



Planning the Reengineering of Legacy Systems

HARRY M. SNEED, *Software Engineering Services*

◆ *How can you know if reengineering is cost-effective? If it is preferable to new development? Or to maintaining the status quo? The author proposes a way to quantify the costs and prove the benefits of reengineering over other alternatives and offers some advice on contracting a reengineering project.*

The success of a large reengineering project depends to a great degree on proper planning. It is not easy to upgrade and migrate several hundred programs and databases without disrupting data-processing service. It is also not easy to justify such a project. Management must be convinced that the organization is really going to achieve a significant benefit in reduced costs and added value. Considering the fact that the functionality of the software remains unchanged, this requires an in-depth analysis of the expected quality and productivity increases. Maintenance metrics are required to measure the improvements.

I have been a programmer for more than 25 years, so I have great respect

for the intellectual effort it takes to develop and prove good software. There is, no doubt, a great need to improve the quality of existing software products and processes. One way to accomplish this is through reengineering. Reengineering poses its own technical challenges — transforming the language, extracting objects, reallocating functions, and proving the equivalence of functionality. However, the overriding business issue is whether reengineering is worth the effort or if legacy systems should be replaced with new ones. It is this question I address here.

PLANNING OVERVIEW

I have developed a five-step reengi-

neering planning process, starting with an analysis of the legacy system and ending with contract negotiation. As Figure 1 shows, the five major steps are:

- ◆ *Project justification.* Justifying the project requires that you analyze the existing software products, the maintenance process, and the business value of the applications. You must project a return on the investment in reengineering — to what degree will the software's quality increase, the maintenance process's efficiency improve, and the business value be enhanced?

- ◆ *Portfolio analysis.* In portfolio analysis, you prioritize applications that are candidates for reengineering according to their technical quality and business value.

- ◆ *Cost estimation.* You estimate the cost of a reengineering project by identifying and weighing all the software components to be reengineered.

- ◆ *Cost-benefit analysis.* In this step, you compare the costs of reengineering with the expected maintenance cost savings and value increases.

- ◆ *Contracting.* The goal of contracting is to reach a maximum level of project distribution, to avoid bottlenecks.

Overcoming this final obstacle means the reengineering project may finally commence, but it will take at least six months of planning to reach this point.

What is reengineering? In planning a reengineering project, I believe it is absolutely essential that you restrict yourself to a one-to-one transformation of business functions and data into different structures, possibly with different languages and a different environment. You must make no functional adaptations or enhancements. If adaptations and enhancements are required, they should be done in a follow-up project. But the content of the databases and system interfaces must not change during the reengineering project or you will not be able to prove the correctness of the transformation. The box on p. 30 defines some of the terms as I

have used them in this article.

The overriding constraint of any reengineering project is: Don't mix *technical* and *functional* reengineering. Technical engineering must precede functional reengineering. There is a pragmatic reason for this rigid separation: It lets you use the same test data to demonstrate the functional equivalence of reengineered software as you used to demonstrate the functional equivalence of the software when it was developed. This is a major distinction from forward-engineering projects, in which an unproven program is tested against hypothetical test data that may or may not mirror the required functionality.

This aspect of having a proven test oracle to test the results against is a major advantage of reengineering. When you alter elementary functions here and there, you lose this advantage

and you may as well rewrite the entire system. This may seem overly restrictive to novices who have never had the responsibility for conducting a large reengineering effort, but it is essential to establishing a measurable baseline.

Objectives. In light of these restrictions, software reengineering has four main objectives:

- ◆ *Improve maintainability.* For example, you may decide to reduce maintenance by reengineering smaller modules with more explicit interfaces. However, it is not easy to measure progress toward this goal. In fact, it takes years to demonstrate a reduction in maintenance effort and even then you cannot prove that reengineering caused the reduction. It may just as easily have been caused by better maintenance staff, better methods, or simply the Hawthorne effect — that

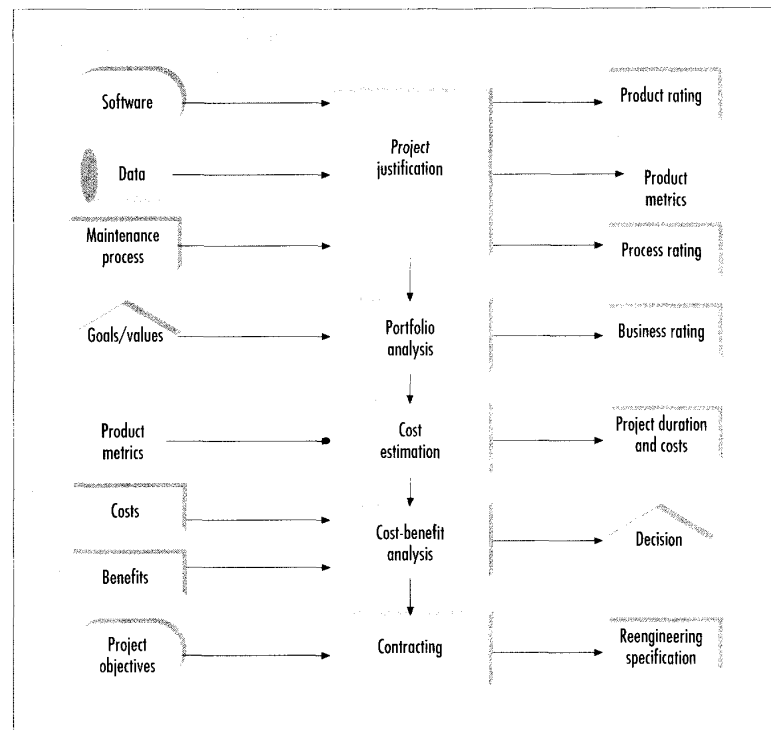
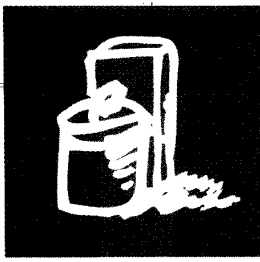


Figure 1. Steps in planning a reengineering project.



which is studied shows improvement. For this reason, it is not easy to justify reengineering projects on the basis of improving maintainability because it requires a major effort to obtain the necessary data.

◆ *Migration.* You may need to move the software to a better or less expensive operational environment; from a mainframe to a Unix server, for example. If the programs are in PL/I they probably must be converted to Cobol or C and this means altering their structure. This goal is easily measured: Either the system performs the same functions in the new environment or it doesn't.

◆ *Achieve greater reliability.* The extensive testing required to demonstrate functional equivalence will unveil old bugs that have been lurking in the software. Restructuring brings

most such potential defects to the surface. This goal can be readily measured by fault analysis.

◆ *Preparation for functional enhancement.* Decomposing programs into smaller modules improves their structure and isolates them from one another, making it easier to change or add functions without affecting other modules. Reengineered, normalized data structures make it easier to add new attributes and relationships. This goal can be assessed only after functional engineering has taken place.

You may combine goals. For example, you could plan to transform programs from a procedural to an object-oriented form to distribute them in a client-server architecture and at the same time plan to reduce maintenance costs by using a language that is more amenable to change.

PROJECT JUSTIFICATION

Justifying a reengineering project is not nearly as easy as many reengineering proponents seem to believe. Managers in user organizations consider software renovation to be the last alternative; development managers and users would prefer to have applications rebuilt from scratch, to fulfill all their suppressed wishes; corporate managers would like to purchase a commercial package; and software maintainers usually prefer to keep things as they are.

In fact, I have found that no one is keen on reengineering. It is a compromise typically reached when new development is too expensive, a commercial package is not available, and the current system is unmaintainable. However, even when reengineering seems the obvious solution, it still must be justified.

Because functionality will remain basically the same, you must prove that reengineering will reduce maintenance costs and increase quality. To do so, you must

- ◆ ascertain current quality,
- ◆ calculate current maintenance costs, and
- ◆ assess the software's current business value.

Attaining these objectives requires a software metrics program; otherwise there is no data available to prove anything. So reengineering justification really has four steps:

- ◆ introduce a measurement program,
- ◆ analyze software quality,
- ◆ analyze maintenance costs, and
- ◆ assess the software's business value.

Measuring maintenance. None of the four main objectives of reengineering can be achieved unless you have quantitative information on the status of your company's data-processing operation. You must know the current maintenance productivity and the system's reliability and maintainability. Without adequate metrics, it is impossible to justify anything. And unless you can

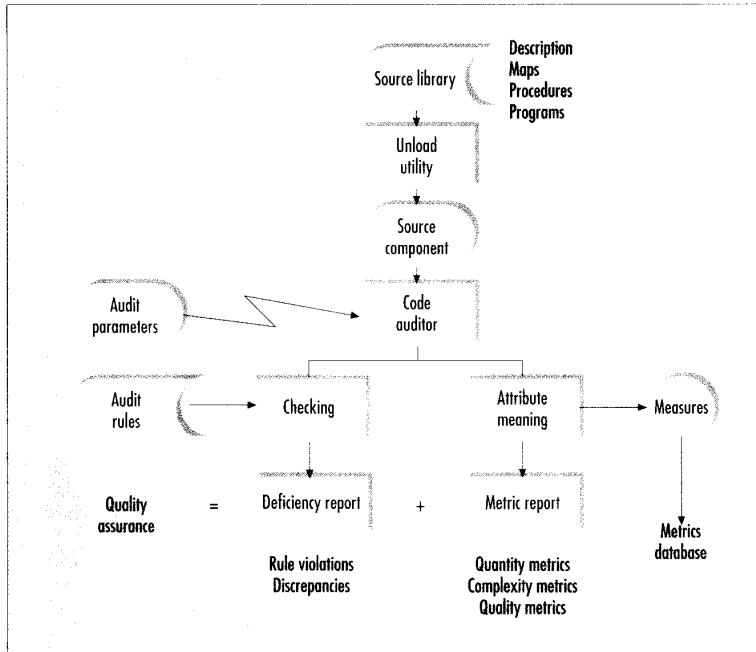


Figure 2. Elements of software-quality measurement. Measurement tools should collect elementary measurements by processing source programs, database structures, job-control procedures, communication interfaces, and data dictionaries. They should also be capable of generating reports on the quantity, quality, and complexity of the code.

quantify the improvements to be gained by reengineering, management will never accept it.

A maintenance measurement program is designed to help you assess the maintainability of the current system, measure maintenance costs, and provide a scale for measuring quality improvement and effort reduction. There must be a measurement program in place before you can justify reengineering costs. From an organizational viewpoint, this means setting up a metrics staff, if possible, in connection with the quality-assurance staff. From a technical viewpoint, it means selecting metrics and installing measurement tools.

Analyzing quality. Figure 2 shows the elements of software-quality measurement. The person responsible for analyzing product quality should be a quality-assurance engineer trained in measurement theory. The measurement tools should collect elementary measurements by processing source programs, database structures, job-control procedures, communication interfaces, and data dictionaries. They should also be capable of generating reports on the quantity, quality, and complexity of the code.

Once a measurement program is in place and the tools installed, you can begin to analyze the legacy system. A code-auditor tool should let you analyze more than 50 programs per person-day. Code auditors use static analysis to extract size, complexity, and quality metrics from the source code. They feed these metrics to a database, where another tool compares them to measures the user would like to have. For example, a user may specify that no procedure should exceed a cyclomatic complexity of 10. Or an organization may forbid `goto` statements because they make it difficult to alter one procedure without affecting the other. Most companies now have such coding conventions. If such quantifiable standards exist, numbers extracted from the source code can be compared

to them and the deviations computed.¹ Several commercial tools can measure cyclomatic complexity, maximum nesting level, degree of module coupling, and distance between variable references. You need only compare these with standardized values. Quality, therefore, is a relative value that is meaningful only when compared to postulated values that are defined by the user or by standards organizations.

Other tools, such as database-schema, map, and report-format auditors, can analyze database structures and communication interfaces. Here, sample measures include the number of keys, the number of relationships among files, the data-dependency rate, and the number of elementary data elements. These measures enhance program measurements, which include lines of code, data elements referenced, database accesses, function points, data complexity, and cyclomatic complexity.

At the end of product measurement, you should be able to assess the maintainability of the software as an aggregated value of the modularity, testability, portability, and complexity ratings of each component. The average value of the individual components of an application, adjusted by the standard deviation, becomes the quality coefficient of that particular application. You use this quality coefficient to assess the need for reengineering and to evaluate the reengineered application.²

Analyzing maintenance costs. You cannot base the decision to reengineer solely on an analysis of the software product itself. In most cases, reengineering involves migrating the product to a more efficient technical environment. This, in turn, entails reengineering the maintenance process itself. So you must also compare the efficiency of the existing maintenance process to

the benefits of the new maintenance environment.

The person responsible for measuring the maintenance process is a metrics engineer, who establishes meaningful relationships between product and process attributes.³ A prerequisite

to this task is the registering, classifying, and auditing of all reengineering tasks. Maintenance tasks are either

- ♦ *corrective* (the analysis and correction of errors; debugging),
- ♦ *adaptive*, (the changing of data formats and algorithms), or
- ♦ *perfective*, (both the optimization and the enhancement of exist-

ing software).

Gerald Berns defined three basic metrics for measuring the maintenance process.⁴ The metrics engineer can apply these metrics to every maintenance task, provided you are auditing each task and collecting measures.

♦ *Impact domain.* This is the percentage of instructions and data elements affected by a given maintenance task, relative to the sum of all instructions and data elements in the system. To measure it, the metrics engineer can use a source-comparator tool to compare the altered software version with the original one. This measure tells you something about the maintainability of the software. The greater this measure becomes, relative to the size of the maintenance operation, the less effective the maintenance process.

♦ *Effort expanded.* This is the number of hours billed by maintenance personnel against the maintenance task, which shows the average number of hours per maintenance task. Your effort-accounting system should distinguish among effort spent on analysis, implementation, and test. The best system is one that presents maintenance personnel with a table of approved maintenance tasks and lets them submit working hours for an

NO ONE IS KEEN ON REENGINEERING. TYPICALLY, IT IS A COMPROMISE. BUT EVEN WHEN IT IS OBVIOUS, IT MUST BE JUSTIFIED.



appropriate phase of each task. The larger this number becomes, the less efficient the maintenance process.

♦ *Second-level error rate.* This is the number of errors caused by maintenance actions. These errors can be picked up both by a regression-testing system that compares the altered test results with the original ones and by an automatic comparison of altered test paths with original test paths.⁵ In both cases, there are expected and unexpected variances. *Expected variances* are those specified by the maintenance task. *Unexpected variances* result from errors in the implementation of the maintenance task. You calculate the error rate relative to the size of the software, the size of the impact domain, and the number of maintenance operations. An increase in this measure indicates decreasing maintenance efficiency.

When all three measurements are increasing, the maintenance process is surely degrading and the time is ripe for renovation.

Assessing business value. The first three steps to justify a reengineering project produce cost indicators. To determine the need for renovation, you must also assess the business value of

the system in question. This is a job for business analysts, who use business-value analysis.

Business-value analysis uses a table like Table 1. The rows list the applications, the columns the business objectives. Typical business objectives are market value, contribution to profit, and the significance of the information the system provides. In each column, the business analyst distributes 100 value points among the applications; those with the greatest value get the highest number of points, and each column sums to 100. The sum of each row becomes the business value of that application relative to other applications. In this way, applications can be ranked according to their relative business values.⁶

An application can also be assessed according to its annual cash value for the company — the total annual revenue accountable, both directly and indirectly, to the use of the data-processing system. For example, the Sabre airline-booking system has direct cash benefits for American Airlines. On the other hand, a reengineered order-entry system may have an indirect cash benefit, because it can process more transactions.

In effect, every business software

system should be booked as a company asset, with an assessed annual value and a depreciation rate that corresponds to its yearly loss of value because of obsolescence and maintenance degradation. *Obsolescence* is a factor of the deviation between what functions the software fulfills and what functions it is required to fulfill. These required functions change from year to year, but fulfilled functions remain constant unless the software is updated. This phenomena, known as the law of continual change, is defined by Manny Lehman and Les Belady.⁷

Maintenance degradation is a factor of rising maintenance costs, longer times to implement the maintenance tasks, growing impact domains of changes made, and increasing numbers of second-level defects. The simultaneous rise of all four metrics is a clear indication of maintenance degradation.

The same value-assessment methods that are applied to applications may also be applied to internal company databases, which have both a relative and an absolute business value. Such an assessment will produce a ranking of databases by volume. If data reengineering is under consideration, you should assess the value of the data separately from the database.

Of course, there will always be business values that cannot be stated in monetary terms. These values may be used to weigh the cash values, such as multiplication factors, but you must be careful not to distort real values with such intangible values.

PORTFOLIO ANALYSIS

A *portfolio analysis* plots applications, according to their need for reengineering, on a chi-square chart like the one shown in Figure 3. The vertical and horizontal axes of the chi square are technical quality and business value.

Technical quality is a weighted mean value of various product and process coefficients selected by the user or imposed by international standards, such as ISO 9126, the new standard for

**TABLE 1
BUSINESS-VALUE ANALYSIS**

Application	Market Value	Contribution to Profit	Information Significance	Score
Credit/Debit	10	10	10	30
Sales Support	40	30	20	90
Stock Inventory	10	30	20	60
Accounting	10	10	30	50
Order Entry	30	20	20	70
	100	100	100	300

product quality. The coefficients are ratios on a scale from 0 to 1. To obtain them, you first define an upper and a lower bound for each quality and productivity metric. The upper bound is the maximum acceptable measure — an error rate of seven defects per 1,000 lines of code or a cyclomatic complexity of 50 per procedure. The lower bound is the maximum attainable quality or productivity value — an error rate of zero defects per 1,000 lines of code or a cyclomatic complexity of 10 per procedure. Then the ratio is

$$1 - \left(\frac{\text{actual_measure} - \text{lower_bound}}{\text{upper_bound} - \text{lower_bound}} \right)$$

So if the actual error rate is three per 1,000 lines of code, the correctness coefficient is 0.572, and if the actual cyclomatic complexity is 22 per procedure, the coefficient is 0.700. I have analyzed hundreds of programs and have never found a conformity ratio above 0.8, which goes to show you that conventions are not worth the paper they are written on unless they are controlled.

The challenge in analyzing technical quality is to set the upper and lower bounds. In my opinion, they must be defined in terms of current industry standards or in terms of what other practitioners are doing. They will vary according to what language is used and they will vary in time as quality and productivity increase.

Business-value analysis is also converted to a comparable coefficient. To derive this, simply divide relative value by absolute value. Thus, if 100 is the maximum value for a certain criteria, and an application scores 30, then its relative business-value coefficient is 0.30. If applications are assessed in terms of their cash value, then divide the annual revenue an application earns by the total annual income of all applications in the same business area.

The higher the technical quality of the application, the higher its plot on the vertical axis. The greater the busi-

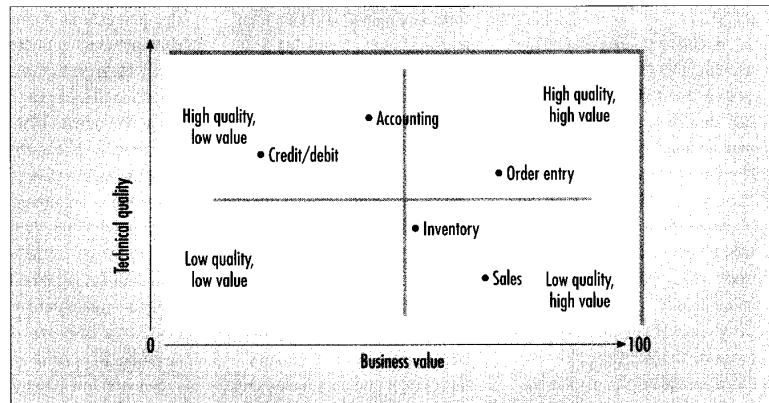


Figure 3. A portfolio plots applications according to their need for reengineering. Those that fall in the lower right quadrant are prime candidates for reengineering.

ness value, the further to the right its plot on the horizontal axis.⁸ Applications whose plots fall in the upper left corner of the square have a relatively low business value and a relatively high technical quality. They do not require reengineering. Applications whose plots are in the lower left corner have both a low business value and a low technical quality. They are candidates for redevelopment or replacement with a commercial package, if one is available. Those whose plots fall in the upper right corner have a relatively high technical quality and a relatively high business value. They may be reengineered, but with a reduced priority. Applications whose plots lie in the lower right corner have a high business value but a relatively low technical quality. These are the primary candidates for reengineering.

COST ESTIMATION

Now the company must consider the cost of each reengineering project. If the costs are higher than the benefits, reengineering is not a viable alternative. The application should be redeveloped or replaced with a commercial package.

Estimating software costs of any

kind is not easy. But for reengineering, it is somewhat more reliable because the application already exists. This means you can count the exact lines of code, the executable statements, the data elements, the database accesses, and the components. These numbers usually increase during reengineering, but at least you have a starting point. Code volume increases during restructuring because the tool must convert tightly coupled networks into loosely coupled trees via node splitting, which leads to redundant code. The only way to reduce the size of the code is to collect redundant operations into common subroutines or place local variables into global areas. Of course, you could also abstract recurring operations of the same type into base classes, but that is pushing the limits of current reengineering technology.

Table 2 shows the calculation of reengineering cost. It is advisable to use the number of noncommented lines to estimate reengineering costs. But this count must be weighted with the various complexity ratings such as cyclomatic complexity, interface complexity, and data-access complexity (which you got from the code auditor in the quality-analysis phase). You can then divide this weighted lines-of-code

REENGINEERING: DEFINITIONS AND OBJECTIVES

Reengineering means different things to different people.

◆ *Business-process reengineering* has to do with business administration. It does not deal with computer processes directly, but with the business processes they support.¹ Software is affected only indirectly, if at all, when the underlying business transactions are altered. In this case, reengineering is only one of several alternatives to implement the changes brought about by business-process reengineering. More often the changes to the business processes are so radical the software must be rewritten. The objective of business-process reengineering is to improve the efficiency of business processes, with or without software.

◆ *Data reengineering* involves restructuring existing databases. The data content remains basically the same, but the form is altered.² For example, data in a hierarchical database that uses pointers to associate records or segments is moved to a relational database that uses keys to associate tuples or rows of tables. In this process, you can alter or enhance the data content, but at some point such a project would cease to be data reengineering and become data engineering with some recycling.

The question is what degree of content change makes reengineering become engineering? Is it five percent? Ten? Fifty? No one can answer this question exactly, so it is better to define reengineering as involving zero-percent

content change. Only the form changes: The same data is migrated from files to databases, from linked lists to normalized tables, or from tables to persistent objects. The access paths are changed, the means of associating access units changes, even the grouping of the data itself may be changed when some data is factored out into new access units. However, the data content remains constant: If there are 512 elementary data items with 1,000 occurrences of each item before restructuring, there are the same number after restructuring. Moreover, the values remain the same even when type and length are altered.

Data reengineering has two main objectives: to improve the data's accessibility and to transfer the data to a supposedly better container. All other objectives — putting data in a normalized form, shortening access paths, providing better query facilities, distributing data among several servers — are subordinate to these two. Even application independence serves to make it easier to access the data from different applications. There are, of course, such goals as data security and space efficiency, but these goals are minor compared with the goals of accessibility and containers. The choice of data container is often the result of a strategic decision made at a high level.

◆ *Software reengineering* involves the renovation of programs and other software artifacts, such as map descriptions, job-control procedures, and data-structure

views. What is true of data reengineering is true of the programs or methods that process the data. That is, in reengineering, the functionality remains basically the same, but the form is altered.³ Elementary operations that were once connected via direct branches in a network are transformed into nodes of a tree that can be invoked only from a superordinate central node. An operation packaged by procedural sequence and triggered by its predecessor is reallocated as a method of a particular data object triggered by an external event.

Usually technical restructuring goes hand-in-hand with a change in the syntax used to describe the operations. For example, assembly is converted to C; C is upgraded to C++; or unstructured Cobol is converted to structured Cobol and from there to Object Cobol. In reality, the language conversion is just a by-product of the underlying structural transformation.

To what degree may a program's functionality be altered or enhanced before it becomes a new program? Five, ten, or 25 percent? Here too, no one can provide a satisfactory answer. It is best to conclude that zero percent is the best way to distinguish reengineering from engineering. If the functionality remains constant and only the form changes, it is reengineering. If even one minor function is added, deleted, or changed, it is engineering. Where else can you draw the line, especially when

drawing up contracts?

◆ *Recycling* differs from reengineering in that the result is not a working system but a collection of reusable components. Recycling is equivalent to stripping parts out of an abandoned automobile or removing organs from a dying patient. It may be economical to reuse parts of a system even though the system as a whole is being abandoned. This objective is quite different from that of reengineering.

The difficulty lies in extracting the components and revising their interfaces so that they are compatible with the other components in the target environment. Recycling may be a way of salvaging past investments without inheriting an overall antiquated business solution, as is the case with reengineering. The greatest advantage of reengineering is that it preserves the existing business solution in a new technical architecture. However, if that solution is inadequate or obsolete, then this advantage becomes a detriment. With recycling the opposite is true: The business solution is abandoned and the technical features are salvaged.

REFERENCES

1. M. Hammer and J. Champy, *Reengineering the Corporation: A Manifesto for Business Revolution*, Harper Business, New York, 1993.
2. P. Aiken, A. Muntz, and R. Richards, "DOD Legacy Systems-Reverse-Engineering Data Requirements," *Comm. ACM*, May 1994.
3. E. Chikofsky and J. Cross, "Reverse Engineering and Design Recovery — A Taxonomy," *IEEE Software*, Jan. 1990, p. 13.

count by the average productivity, measured in lines of code per month. For average monthly productivity, use measures from previous reengineering projects — for example, 20,000 lines of Cobol per person-month. This will give you the number of person-months reengineering will cost. These are the reengineering costs, minus testing.

To estimate testing costs, first count the number of test cases required to meet the minimum test-coverage requirements. Multiply this by the average cost of a test case. The number of test cases required can be induced from the cyclomatic complexity of the programs.⁹ This implies that the programs are subjected to a path analysis. This result should then be weighted by such factors as testability, test support, and test environment. As a rule, test costs will be at least as much as the reengineering costs, if not double or even triple. Reengineering itself is highly automated, but testing is not.¹⁰

To estimate effort for data reengineering, count the number of databases, files, and fields. Weigh this sum by the data-structure complexity and degree of data independence. Then divide the result by the monthly productivity rate in data elements.

If you are to reengineer job-control procedures, first measure them in terms of the number of program steps and the number of file or database allocations. Measure user interfaces in terms of the number of transaction types and the number of maps, windows, and reports involved.

Generate a productivity table that converts the number of each software item into person-days of effort.

When you have calculated the total person-days of effort, you can derive the minimum project duration using a variant of Cocomo,¹¹ adjusted for the increased parallelism of reengineering projects. On the basis of my experience, the thousand-lines-of-code count used in Cocomo to estimate project effort must be divided by at least two for assembly, three for PL/I and C,

TABLE 2
CALCULATING REENGINEERING COSTS

Program	Noncommented Lines of Code	Complexity Factor	Adjusted Lines of Code
SBG TP 01	2,054	0.85	1,746
SBG TP 02	2,930	0.92	2,969
SBG TP 03	2,588	1.02	2,640
SBG TP 01	3,360	1.15	3,864
SBG TP 02	3,120	0.88	2,746
SBG TP 03	3,650	1.12	4,088
•	•	•	•
•	•	•	•
•	•	•	•
System	52,250	1.05	54,862

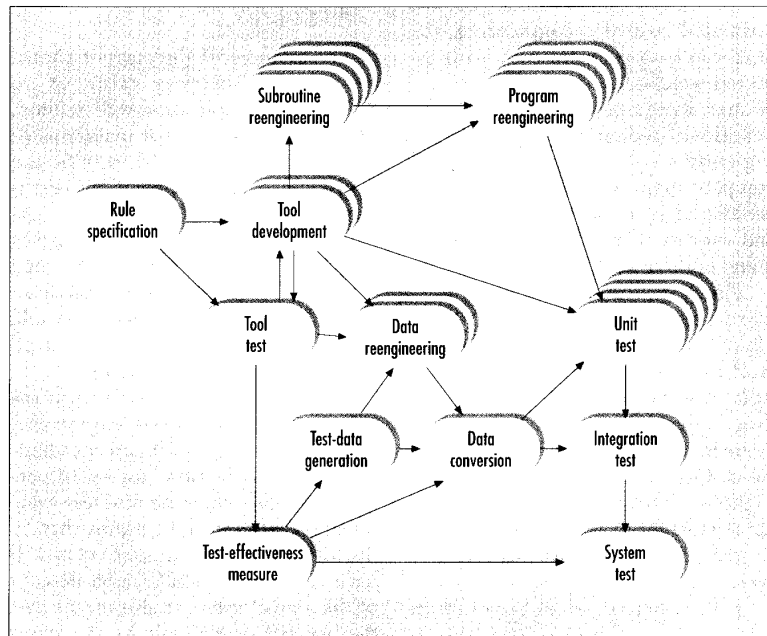


Figure 4. Reengineering tasks are less interdependent than new-development tasks, so some of the work can be done in parallel.



and four for Cobol.

To estimate project duration, we adjust the Cocomo equation

$$T_{dev} = 2.5 (\text{effort})^{0.38}$$

to

$$T_{dev} = 2.5 (\text{effort})^{0.19}$$

As Figure 4 shows, many tasks such as data reengineering, program reengineering, and job reengineering are less interdependent compared with development projects, so more jobs can be done in parallel. To make it easier to do tasks in parallel, create simulated test environments for each reengineering task type. My company, for example uses a reengineering workbench to simulate a host database and teleprocessing environment and to generate data from the original program. We also validate the reengineered databases separately before integrating them into the target environment.

COST-BENEFIT ANALYSIS

Now that you have calculated the estimated cost of reengineering, the next step is to compare costs with expected benefits. It is not enough to examine the benefits of reengineering only — you must compare them with the benefits of redeveloping and with the benefits of doing nothing at all.

These 16 parameters were selected by the Union Bank of Switzerland in 1990 to determine whether to abandon, redevelop, or reengineer legacy applications. Originally proposed by Robert Figlio in 1989,¹² I have refined these parameters for reengineering:¹³

- ◆ P_1 : Current annual maintenance cost.
- ◆ P_2 : Current annual operations cost.
- ◆ P_3 : Current annual business value.

◆ 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_{10} : Predicted annual maintenance cost after redevelopment.

◆ P_{11} : Predicted annual operations cost after redevelopment.

◆ P_{12} : Predicted annual business value of the new system.

◆ P_{13} : Estimated redevelopment costs.

◆ P_{14} : Estimated redevelopment calendar time.

◆ P_{15} : Redevelopment risk factor.

◆ P_{16} : Expected life of the system.

The benefit of maintaining the status quo — that is, continual maintenance without reengineering, is given by the formula

$$\text{Benefit}_M = [P_3 - (P_1 + P_2)] * P_{16}$$

For example, if the current annual business value of an application is \$1 million, the annual maintenance costs is \$250,000, the annual operations cost is \$250,000, and the expected life of the legacy system is five years, then the benefit of maintaining the status quo would be \$ 2.5 million, if all remains constant.

But what if maintenance costs are increasing by 20 percent annually and the operation cost by 10 percent annually, while the business value remains constant. This means that by the fifth year, the annual costs will have reached \$884,025 or 88 percent of the annual value. It also means that the benefits of maintaining the status quo for another five years will only be \$1.67 million. In this case, we should

enhance the formula by the cost and value increases

$$\text{Benefit} = \{ P_3 + P_3 * V_i \}_{1...P_{16}} - \{ (P_1 + P_1 * M_i) \}_{1...P_{16}} + \{ 2 + P_2 * O_i \}_{1...P_{16}}$$

where V_i is the annual value increase; M_i the annual maintenance cost increase; $1...P_{16}$ is the number of years remaining in the planning period, and O_i the annual operation cost increase.

The benefit of redeveloping the system is calculated as

$$\text{Benefit}_D = [(P_{12} - (P_{10} + P_{11})) * (P_{16} - P_{14}) - (P_{13} * P_{15})] - \text{Benefit}_M$$

For example, if the predicted annual business value of the new system is \$1.5 million, the predicted annual maintenance costs \$150,000, the predicted annual operations costs \$100,000, the estimated development costs \$1.5 million, the risk factor 1.2, and the estimated time of development two years, then the benefit of redevelopment would be \$1,950,000 – \$ 1,610,000 = \$340,000. The increase in the business value is offset by the development costs and the short remaining life span of only three years, leaving us with only a marginal benefit.

Redevelopment may extend the application's life. If it increases the life span by just one more year, the benefit of redevelopment rises significantly, because development costs will have been recovered in the fifth year. The total costs of maintaining the old system for six years will be \$4,401,903; whereas, the total value of the old system over the six-year period will be \$6 million, leaving a benefit of \$1.6 million. The new system will accumulate a four-year value (six minus two years for development) of \$6 million. When the four-year cost of \$1 million is deducted, \$5 million is left. Subtracting the development cost of \$1.8 million leaves an absolute benefit of \$3.2 million and a relative benefit of \$1.6 million compared with the first alternative. Thus, in the long run, it may always be better to redevelop if you

**YOU MUST
COMPARE THE
BENEFITS OF
REENGINEERING,
REDEVELOPING,
AND DOING
NOTHING AT ALL.**

have the capital and resources to do so.

You can calculate the benefit of reengineering the system in a similar fashion.

$$\text{Benefit}_R = [(P_6 - (P_4 + P_5)) * (P_{16} - P_8) - (P_7 * P_9)] - \text{Benefit}_M$$

For example, if the predicted annual business value of the reengineered system is \$1.1 million, the predicted annual maintenance cost after reengineering is \$200,000, the predicted annual operations costs \$100,000 (the same as a new system), the estimated reengineering costs \$500,000, the risk factor is 1.0, and the estimated time of reengineering is one year, the benefit of reengineering would be \$2,700,000 - \$1,610,000 = \$1,090,000.

This is \$750,000 more than the redevelopment value after five years. The fact that reengineering can be done for half the cost and in half the time outweighs the increased business value of the redeveloped system in light of the system's short life span. If the life span were 10 years instead of five, the result would be very different, because the annual value of the redeveloped system after costs is \$1.25 million as compared with the \$800,000 value of the annual reengineered system after costs. With another two years of life, the redevelopment would be amortized. Two years less and it would be better to do nothing at all. Therefore, the predicted life span is critical. Whether or not you choose to reengineer depends on your planning time frame.

Of course you must take other factors into consideration. It could be that the one year difference between the time the reengineered system is available and the time the redeveloped system is available is so critical that it doesn't matter how long the system lives. Other factors are user satisfaction and maintenance programmer morale. It could be that an organization is about to lose its last maintenance programmer if it doesn't migrate to a more convenient environ-

ment. Or it is about to lose its last customer if it doesn't stabilize the system. In such cases, cost-benefit calculations are only eyewash — the user organization must choose the quickest solution. This is *emergency reengineering*: Getting the software onto a more modern environment and improving the technical quality with a minimum of change.

CONTRACTING

There are two possible kinds of reengineering contracts, by time and material and by result (turnkey). In the case of development projects, time-and-material contracts are the most common because it is difficult to specify all tasks and estimate their costs. Only when development projects have been specified to a very fine degree do they lend themselves to turnkey contracts.

Reengineering projects, on the other hand, lend themselves very well to turnkey contracts because the components are known, and their size, complexity, and quality can be measured.

Task definition. You can derive a list of tasks to be performed from the objects that must be reengineered. In general, each software object is going to require restructuring, conversion, testing, and postdocumentation. Each group of objects will require integration testing and process documentation. And the system as a whole will require system testing, configuration management, and system documentation.

Don't overlook the task of preparing test data. Another advantage reengineering has over new development is that reengineered programs can be tested against existing test data. In fact, the existing test data is a de facto specification. However, if the existing set of test data is inadequate or

not representative, preparing test data can become a significant task. To determine if the test data is adequate, measure the testing of existing programs. This adds two more tasks, one to create the test data and one to measure test effectiveness.

These testing tasks should be done by the users because they are equivalent to specifying the application and so are the baseline for whether or not the reengineered system will be accepted. If these testing tasks are contracted out, it must be done on a time-and-material basis because, like new-system specification, they are highly user-dependent.

The same applies to the tasks at the end of the project. System testing and configuration management must be the responsibility of the user, preferably the future maintenance programmers. Through system testing, configuration management, and system documentation, they will become familiar with the internals of the reengineered software.

All the other tasks can be contracted out on a fixed-price basis. It's probably best to base payment on lines of original source code. In my experience, prices range from 50 cents for third-generation code to \$4 for assembly code. You may distinguish between procedural statements, data declarations, and other lines. And be sure to include machine use and unit test in the price.

Effort distribution. To maximize effort distribution, I recommend that you perform reengineering and unit testing on desktop workstations. Integration and system testing will, of course, have to take place on the target machine, but that should be all. The objective of the contract should

IT IS ALL TOO EASY TO GET SIDETRACKED INTO SOLVING TECHNICAL ISSUES AND TO FORGET WHY YOU ARE DOING THE WORK IN THE FIRST PLACE.

be to perform as much of the reengineering work as possible independent of the target machine. Not only does this avoid bottlenecks in machine capacity, but it also allows a maximum work distribution, thus reducing the project duration. Time is critical in reengineering projects.

You must compensate for the communication problems that can result from such wide project distribution with strict project standards, well-defined acceptance criteria, and the use of common tools. You must state clearly in the contract which programming conventions are to be met and how the unit test is to be measured. For example, you might state that each program must obtain a 95-percent conformance rate and a 90-percent test coverage rate. The conformance rate will be measured by a static analyzer, the test coverage by a dynamic analyzer. Furthermore, the test results of the reengineered program must correspond 100 percent to the test results of the original program.

Such precise contractual obliga-

tions make it clear to all involved what must be accomplished. It is up to the contracted party to deliver the results. How they do this is their business. There are several automated systems available on the market to support the reengineering process, such as Refine/Cobol from Reasoning Systems and Recoder from Knowledgeware. It is beyond the scope of this article to review and compare them. Besides, this is a matter for the subcontractor. If the tools are very good, costs will be lower. For the customer, tools are only important insofar as they affect the quality of the reengineered product. The software could be reengineered manually in India or automatically in Germany. The only important thing is that the acceptance criteria are met.

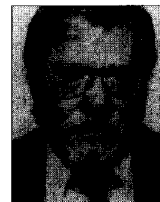
It is not easy to sell reengineering. As the manager of a small software-reengineering company, I am continually confronted with the task of justifying reengineering. The user wants to know what the benefits are. Why

reengineer old Cobol or Fortran when there are so many attractive fourth-generation and object-oriented languages on the market?

That is why I have chosen to address business issues — not because the technical problems of transforming code and data structures are not important but because they may be irrelevant if you are not able to make a business case for solving them. There is nothing worse for a technician than to be working on a solution to some problem for years, only to discover that the problem was incorrectly stated from the beginning. This danger is highly eminent in software reengineering. It is very easy to get sidetracked into solving local technical issues and to lose the global perspective as to why you are doing this in the first place. I hope I have reminded engineers of the business environment in which they are working and reminded reengineers of the fact that reengineering is only one of many solutions to the maintenance problem. ♦

REFERENCES

1. D. Coleman et al., "Using Metrics to Evaluate Software System Maintainability," *Computer*, Aug. 1994, pp. 44-49.
2. D. Rombach and V. Basili, "Quantitative Software," *Qualitätssicherung Informatik-Spektrum*, No. 10, 1987, pp. 145-156 (in German).
3. N. Fenton, *Software Metrics — A Rigorous Approach*, Chapman & Hall, London, 1991.
4. G. Berns, "Assessing Software Maintainability," *Comm. ACM*, Jan. 1984, pp. 32-49.
5. J. Hartman and D.J. Robson, "Revalidation During the Software Maintenance Phase," *Proc. Conf. Software Maintenance*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 70-79.
6. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, Reading, Mass., 1988.
7. M. Lehman and L. Belady, *Program Evolution — Process of Software Change*, Academic Press, London, 1985.
8. G. Verdugo, "Portfolio Analysis — Managing Software as an Asset," *Proc. Int'l Conf. Software Maintenance Management*, Software Maintenance Assoc., New York, 1988, pp. 17-24.
9. J.R. Horgan, S. London, and M. Lyu, "Achieving Software Quality with Testing Coverage Measures," *Computer*, Sept. 1994, pp. 60-69.
10. H. Sneed, "Regression Testing of Reengineered Software," *Proc. Conf. on Program Comprehension*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 149-155.
11. B. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J., 1983.
12. R. Figlio, "Benefits of Reengineering," *Proc. Software Maintenance Assoc. Conf.*, Software Maintenance Assoc., New York, 1989.
13. H. Sneed, *Software-Wartung*, Rudolf Mueller Verlag, Cologne, Germany, 1990.



Harry M. Sneed is technical director at Software Engineering Service, Munich, where he is responsible for reengineering tools and projects at the Union Bank of Switzerland, Zurich. He has developed tools and techniques for software reverse engineering and reengineering since 1980, and has planned and undertaken several large reengineering projects for firms in Germany, Austria, and Switzerland since 1987. He has written seven books and more than 50 technical articles on various aspects of software engineering.

Sneed received an MS in public administration and information sciences from the University of Maryland at College Park. He is a member of the IEEE and ACM.

Address questions about this article to Sneed at SES, Rosenheimer Landstabe 37, D-85521 Ottobrunn, Munich; 100276.57@compuserve.com.