NIST Special Publication 500-193

Computer Systems Technology

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

NIST

Software Reengineering: A Case Study and Lessons Learned

Mary K. Ruhl and Mary T. Gunn

NIST Special Publication 500-193

Software Reengineering: A Case Study and Lessons Learned

Mary K. Ruhl and Mary T. Gunn

Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

September 1991

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
John W. Lyons, Director

Reports on Computer Systems Technology

National Institute of Standards and Technology Special Publication 500-193
Natl. Inst. Stand. Technol. Spec. Publ. 500-193, 39 pages (Sept. 1991)
CODEN: NSPUE2

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1991

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402

Abstract

This report is aimed at managers and technical personnel (both Federal Government and industry) who need to understand:

- the concepts and issues of software reengineering,
- the use of Computer-Aided Software Engineering (CASE) tools in the reengineering process,
- and the application of this technology to organizational problems.

Software reengineering involves the use of existing software and documentation to specify requirements, design, documentation, and to produce software for a target platform. CASE tools are expected to play an important role in automating parts of the reengineering process.

In this report software reengineering and other related terms are defined and possible benefits that relate to this technology are described. The use of CASE tools for reengineering are examined. A case study that examines the feasibility and cost-effectiveness of software reengineering is described. Study results are addressed along with recommendations for organizations that are considering the use of reengineering.

Keywords

CASE (Computer-Aided Software Engineering) tools, design recovery, reengineering strategies, reverse engineering, software reengineering.

Acknowledgments

Special thanks to Wayne McCoy and Bruce Rosen of NIST for their valuable comments and guidance. We would also like to thank our reviewers: Dolores Wallace, and Neva Carlson of NIST; George Baird and Dr. Paul Oliver of Booz, Allen, and Hamilton Inc.; and Dolores Martinez, Cathy McKinney, Dan Scott, and George Wall of IRS.

Executive Summary

Software reengineering involves the use of existing software and documentation to specify requirements, design, documentation, and to produce software for a target platform. Many Federal government agencies and other organizations are evaluating the migration of older software to more powerful, more open computing environments. Additional system concerns include the high cost of software maintenance, the need to gain a better understanding of existing systems, and the impact of reduced computer systems budgets. Federal agencies are looking to software reengineering as a solution to these problems.

A case study conducted by the National Institute of Standards and Technology (NIST) and the Internal Revenue Service (IRS) indicates that software reengineering can be a cost-effective, viable solution for extending the lifetime of an application system. The degree to which it is cost-effective depends on the goals for reengineering, the condition of the original application system and documentation, available automated tool support, and the involved personnel.

The context for reengineering should be established in terms of the corporate goals for the organization before undertaking the task of reengineering. It is also important to clearly define the system goals and motivations for reengineering. Clearly defined goals are needed to determine a suitable approach for reengineering.

A variety of approaches can be employed to gain the benefits of reengineering. These approaches differ by the amount of design that is to be retained from the original system, the organization's reengineering goals, the condition of the current system, and the resources to be allocated to the project. Before determining a reengineering approach, the application system should undergo a thorough evaluation to determine what is worth retaining for future use and what is not. During the evaluation, data definitions and usage, code, documentation, maintenance history, and appropriate metrics should be analyzed to determine the current condition of the system.

The case study indicates that full support for software reengineering from CASE tools is currently lacking in several aspects. Most currently available CASE tools are directed at one particular aspect of software reengineering and are targeted for a certain environment. Therefore, expectations for automated support from CASE tools must be realistic. Provisions in terms of personnel, effort and tools must be made to compensate for the lack of full support of the reengineering process by currently available off-the-shelf tools.

Performing reengineering requires a highly trained staff with experience in the current and target system, the automated tools, and the specific programming languages involved. Application system experts must be involved throughout the reengineering process; they are essential for design recovery.

Software reengineering is a complex and difficult process. The success of an organization's application of this technology will be determined by the level of commitment made by the organization.

Table of Contents

1. Introduction

Many federal agencies are faced today with the problem of operating and maintaining obsolete software and hardware. Gains in microprocessor, operating system, and communication technologies have enabled faster and more flexible computing than that which was possible when many of today's operational government systems were created. To achieve improvements in system operation and performance while protecting their software investment, many organizations want to migrate existing software to new computing platforms.

The high cost of software maintenance (enhancement, adaptation to new environments, and error correction) is another problem facing many organizations. Usage of disciplined design, implementation, testing and maintenance methodologies helps to ease the maintenance costs, but estimates for software maintenance are still high — consisting of 60 to 80% of today's total software cost.

The documentation of a software system is rarely up-to-date. Changes to the code are typically made without a corresponding change to the documentation. Inconsistency between documentation and code can make software maintenance difficult. In addition, the documentation may be incomplete. Unrecorded information may be known only to those who deal with the system on a daily basis and this information can become lost over the years due to personnel changes. Inconsistent and incomplete documentation complicates and lengthens the maintenance process and cause a dependence on certain system personnel for operation.

In the 1960s, 80% of the total system cost (amount expended over the lifetime of the system) was appropriated for hardware and 20% for software. In 1985 these estimates had been reversed, with software consuming 80% of the system budget [FAIR85]. The reasons behind this trend are the dramatic decrease in hardware costs (due to advances in semiconductor fabrication technology), the labor-intensive nature of software, and the increase in personnel costs. The proportion of expenditure on software emphasizes the importance of structure and maintainability of software, and demands that software have a long lifetime.

There is another economic factor driving the use of software reengineering. Cuts in government and industry budgets have reduced the resources available for the development of new software systems. Adapting suitable systems to include new functionality can lead to cost savings for organizations.

To address these problems and others, a number of federal agencies are looking to software reengineering, which is directed at extending the lifetime of software. Reengineering enables the salvaging of past work for future use, thus retaining the software's value to the organization.

In this report software reengineering and other related terms are defined and the possible benefits that can be gained from this technology are described. The use of CASE tools for reengineering and various issues related to the application of CASE technology to reengineering are discussed. A case study that examines the feasibility and cost-effectiveness of software reengineering is described. Study results are discussed along with recommendations for organizations that are considering the use of reengineering.

Various market products will be mentioned for clarification purposes only.[1]

2. Software Reengineering and CASE Technology

2.1 Definitions and Related Terms

The term "software reengineering" and related terms currently mean rather different things to different people. This is the cause of much confusion in this area. Elliot Chikofsky and James Cross II provided specific definitions for many terms related to reengineering in [CHIK90]. For the purposes of this document, the definitions of reengineering and related terms that Chikofsky and Cross have defined will be used.

Forward engineering is "the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system" [CHIK90].

Reverse engineering is "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [CHIK90]. It is the derivation of system design specifications based on the physical system description. This involves analyzing the code, all documentation, and recording relevant information from the human users and maintainers. Reverse engineering does not involve modification to the system, only examination of the system. Also, it is not necessary to start reverse engineering at the lowest level description (code) — it may be started at a higher level (such as design). When abstracting low-level information to higher-level descriptions, optimization mechanisms or environment dependencies necessary for the original environment are removed. Those mechanisms that are applicable for the target environment can be added during forward engineering.

There are a number of sub-areas of reverse engineering, two of which are redocumentation and design recovery. *Redocumentation* is "the creation or revision of a semantically equivalent representation within the same relative abstraction level" [CHIK90]. *Design recovery* is undertaken when "domain knowledge, external information, and deduction of fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself" [CHIK90]. Design recovery is essentially combining all information pertinent to a system —

[1]NIST does not recommend or endorse products and nothing herein is intended as an endo rsement of any product.

what it does, how it does it, why it does it, etc. This area of software reengineering is the most difficult part since it deals with such a wide range of information, some of which is not easily attainable.

Software reengineering is defined as "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form" [CHIK90]. Software reengineering consists of reverse engineering and then some form of forward engineering or modification. Enhancements to meet new requirements that were not in the original system may subsequently be performed.

Another related term is *restructuring* which is "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (function and semantics)" [CHIK90]. Code restructuring is often performed on poorly structured code in order to make it more maintainable. Today there are automated utilities available that, with the guidance of a human analyst, will alter code so the rules of structured programming are closely followed. For example, GOTOs are removed or software modules are modified to ensure one entry and one exit.

Reengineering a system involves not only the process or procedure side, but also the data side of an application system. *Process reengineering* is a code-level procedure that analyzes control flow. A program is examined to create an overview architecture with the purpose of transforming undesirable programming constructs into more efficient ones. Restructuring can play a major role in process reengineering. *Data reengineering* examines and alters the data definitions, values and the use of data. Data definitions and flows are tracked through the system, and through this process, may reveal hidden data models. Data names and definitions are examined and made consistent. Hard-coded parameters which are subject to change may be removed [RICK89]. This process is important because data problems are typically deeply rooted within systems.

Many of the steps involved with system reengineering are being addressed through the application of Computer-Aided Software Engineering (*CASE*) Technology. CASE technology is the automation of the software development processes. It automates many analysis and design tasks, thus seeking to increase productivity in the areas of software development, implementation and maintenance. CASE technology is a combination of development methodologies and automated tools.

2.2 Motivations For Reengineering

There are many reasons why an organization might consider the use of software reengineering. The primary motivational factor is the possible cost savings from the use of this technology. Cuts in government and industry budgets have reduced the resources available for the development of new software systems. Consequently, organizations are seeking ways to adapt current systems to accommodate new functionality to meet changing needs. Reengineering software essentially salvages the past work for future use, thus retaining the software's value to the organization.

One reason to consider reengineering is possible reduction of maintenance costs. In many of today's systems, maintenance changes have been directly implemented in the code and have not been carried back to the design documentation of the systems. Lack of documentation and complexity of code force maintainers to devote an extensive amount of time trying to understand the functions of a system. Reengineering provides a means of reworking the documentation and code into a more maintainable format that allows maintainers to quickly gain a better understanding of the system.

It is not necessary to apply the entire reengineering process to achieve the benefit of reducing maintenance costs — accomplishing part of the process, such as design recovery and restructuring, can have a significant impact on maintenance costs. Design recovery, for example, can be performed to recover and record lost system information. This can reduce an organization's dependence on those individuals who understand the present software, and shorten the time necessary for new individuals to learn the system.

Reverse engineering can be used to gain a better understanding of the current system's complexity and functionality, and to identify "trouble-spots." Errors can be detected and corrected, or modifications made to improve system performance. The information gained during reverse engineering can be used to restructure the system, thus making the system more maintainable. Maintenance requests can then be accomplished more easily and quickly.

Another area in which it is useful to consider the use of software reengineering is that of migrating the functionality of an older system to a new computing environment. This is especially true in those cases where it is necessary to enhance the system to satisfy new requirements. By reverse engineering, information for the development of the application on a new environment is collected. An analysis of current functionality in light of new system requirements may permit redesign of the system. The software can then be forward engineered to the target environment directly from the gathered information (if the information is sufficient).

Software reengineering also enables the reuse of software components from existing systems. The knowledge gained from reverse engineering can be used to identify candidate systems composed of functions (reusable components), which can then be used in other applications. Reverse engineering can also be used to identify functionally redundant parts in existing applications.

2.3 The Use of CASE Tools for Reengineering

Computer-Aided Software Engineering (CASE) tools are automated tools that organize, structure, and simplify the software life-cycle. The automation of tedious software engineering tasks by CASE tools has enabled better control of software development and its management. A broad range of CASE tool products has emerged including planning, designing and modeling tools, as well as code generators and some reverse engineering tools. Users are now considering CASE tools to automate the reengineering process.

At the present time, many software engineering tools are being developed and marketed for business systems environments. Tools and methodologies for technical scientific environments are minimal [HAUG91]. The business system tools carry out the functions of reengineering at varying levels, with a majority concentrating on the forward engineering process. The typical CASE tool automates only a portion of the software life cycle. More than 200 CASE vendors support the conventional forward engineering process, but very few tackle the difficult problems of reverse engineering [MART90b].

Currently, the available reverse engineering products are typically analyzers that examine the structure of source code and generate a more abstract specification, like pseudo-code or structure charts. Such tools are oriented towards a particular set of machines or environment (e.g., mainframe COBOL applications, multi-user LAN applications). Available CASE tools do not implement the full scope of design recovery; abstraction based on the recognition of certain structures is currently achievable. Analysis by humans is essential for identifying what information is important, determining the functionality of each program and the entire system, and judging whether the functionality is necessary. Also, some information will probably not be in a format recognizable to an automated tool. It is doubtful that design recovery can be fully automated because of the human judgment element needed for observation and fuzzy reasoning.

Tools today support a variety of development methodologies. Before reengineering a system, it is important to identify the methodologies and life-cycle stages supported by a tool or set of tools. Some tools will enforce adherence to a strict development methodology whereas other tools enforce no methodology. Reengineering may cause drastic changes to the environment and methodology of an application system which could lead to frustration for individuals currently working with the system. The tools chosen for reengineering should support methodologies which are applicable to the requirements of a particular organization.

2.4 The Repository

The typical reengineering environment consists of multiple tools and a repository that serves as the focal point for all development activities. The repository serves as a system information resource across applications and tools for the entire system life-cycle. Reengineering tools are used for analyzing a subject system and storing relevant information gathered during analysis in the repository. If the stored information is sufficient, a new system could be forward engineered directly from knowledge contained in the repository.

Reengineering a system may require the use of multiple tools developed by different vendors. The use of multiple tools raises concerns with how well the tools handle data interchange. Data interchange is the transfer of data among different CASE tools and repositories.

Typically each repository will handle the storage and definition of data structures that support data in a different manner. These differences make data interchange extremely difficult or costly. In addition, it is common to suffer a loss of data when transferring data among repositories. It is critical that in the future CASE tools provide a standard method and functionality for transferring data between tools and repositories.

When using automated tools for reengineering, consideration should be given to how well the tools handle data integration. Data integration is the unification of data used by an entire organization. This allows an organization to improve productivity by utilizing the same data across all business systems. For data integration to succeed, standard data administration policies and practices must be enforced by the organization. Integrated software engineering tools and repositories should assist an organization in establishing and enforcing data administration policies which regulate the definition and structure of data.

There are several standardization efforts underway that are significant to repository. Both the American National Standards Institute (ANSI) and International Organization for Standardization (ISO) are working on a standard for Information Resource Dictionary System (IRDS), which is essentially a set of software specifications for a standard data dictionary system [ROSE89]. The ANSI standard was approved in 1988 (X3.138-1988) and was adopted as FIPS 156. ISO SC21 IRDS Working Group is developing a different Services Interface, to be used for communication between the IRDS and other relational model based software (e.g., DBMS).

Additional standards efforts are focusing on the development of an integrated CASE environment allowing for the integration of different tools to support various life cycle development activities. Working groups have been formed within the Institute of Electrical and Electronics Engineers (IEEE), ISO, ANSI, and other industry organizations to develop standards for tool integration, tool to tool information exchange, open architectures and portability [SHAR91]. The repository will serve as a key component within these standards activities, providing a facility for the storage, maintenance, and exchange of data.

NIST has established a program to develop a Reference Model (RM) for an Integrated Software Engineering Environment (ISEE). The ISEE RM will provide a framework for tool integration. The European Computer Manufacturers Association (ECMA) Reference Model has been adopted as a base document. An anticipated use of the RM will be to identify the areas in which relevant standards currently exist or need to be developed to support integrated CASE.

3. A Government Case Study

3.1 Background and Goals

In order to investigate the feasibility and cost-effectiveness of reengineering existing code, a case study was performed in which a structured COBOL application system was reengineered and migrated to a more disciplined, more open environment. The application system was provided by the Internal Revenue Service (IRS). The purpose of the case study was to evaluate the applicability of reengineering technology for use in the Federal Government. NIST conducted a competitive procurement to award a Labor Hours contract for performance of the case study. The ceiling price for the contract was set at $250,000.

It is important to note that this is only one case study conducted on one application system, using a particular set of tools and reengineering methodology. Also, the approach was based on a certain set of goals for the study. Additionally, this case study focused on a business oriented application. Reengineering military applications with demanding real-time constraints and embedded assembly code would require different tools than those used in this study. Different goals, application systems, CASE tool selections, and methodologies may have very different results. Thus, the results documented in this publication neither recommend nor condemn the CASE tools or employed practices of the contractor selected. Our hope is that, despite such differences, other organizations can apply lessons from this case study to aid in determining an appropriate approach for their organization.

The reengineering project was conducted on the IRS Centralized Scheduling Program (CSP) system. This system was written in 1983 using structured COBOL 74 for Unisys 1100 hardware. It includes batch jobs, database queries and updates, and on-line processing. The application system is made up of 37 source programs consisting of approximately 50,000 lines of COBOL code, along with 53 subroutines of MASM assembly language, consisting of 2,738 lines. The database is a DMS 1100 network database. The application system is currently in use at the IRS and is one of several currently operating on the Unisys 1100 which serves approximately 1000 users. Documentation for the original application system that was provided to the contractor included Data Flow Diagrams (DFDs), Functional Specification Packages (FSPs — in a structured English format), Computer Programmer Books (CPBs), relevant schema definitions from the DMS 1100 database, and Nassi-Schneiderman diagrams. All relevant documentation that was available within IRS was provided.

The contractor was to examine the effort needed and issues by attempting to reengineer the CSP system to a more open target environment and convert the network data base to a relational database, normalized to Third Normal Form (3NF). The target environment and reengineering tools were to be chosen by the contractor. Selections were to address the Government's interest in the extent to which Federal Information Processing Standards (FIPS) are included or considered. Particular standards of interests for this study included SQL (FIPS PUB 127), Portable Operating System Interface (POSIX, FIPS PUB 151), Government

Open Systems Interconnection Profile (GOSIP, FIPS PUB 146) and IRDS (FIPS PUB 156). It was realized that implementations do not currently exist for some of these standards. Therefore, an evolutionary path to these standards was to be shown for the proposed target environment. Additionally, the use of custom tools was to be limited — off-the-shelf tools were preferred. The contractor was required to demonstrate that the reengineered system was equivalent to or better than the original in behavior, outputs, and performance. Application system experts for the current system were on-site throughout the project to provide expertise on system operation. IRS and NIST personnel were on-site to receive training in the methodology and tools.

It was assumed that for software reengineering to be considered cost-effective, the process should, in principle, be achievable in a fairly short period of time. Accordingly, an aggressive 17 week time schedule was set for the project. Following a competitive procurement, the contract was awarded to Booz, Allen & Hamilton Inc. of Bethesda, Maryland.

3.2 Technical Approach

Two major off-the-shelf tools were selected for this project, one to support reengineering the data side of the application system, and the other to support reengineering the process side. Each of these tools has high visibility for its functionality and a fairly large market base (as compared to other tools). Additional tools (some proprietary) were used to analyze COBOL procedure division code, develop higher-level design documentation, and produce metrics data from the COBOL source programs. The need for some of the additional tools was not apparent at the start of the project. As the project progressed and difficulties were encountered with some off-the-shelf tools, on-hand, proprietary tools were utilized as a solution.

The reengineering methodology was broken down into the five steps listed below:

Step 1: baseline the original system;

Step 2: extract/analyze data, code functionality and documentation;

Step 3: produce documentation;

Step 4: generate new code;

Step 5: execute and test code.

In the context of the defined terms of section 2.1 of this document, reverse engineering is accomplished in Steps 1 through 3 while forward engineering is completed in Step 4.

Portions of the system were identified and prioritized for reengineering. Programs were categorized by level of complexity and the interfaces between programs. The categories, in order of increasing complexity, were:

- batch programs not accessing the database, not using COBOL's Report Writer or Sort;
- batch programs accessing the database;
- batch programs using Report Writer;
- interactive programs (included database access and screen interface).

3.3 Issues

Difficulties were encountered during the reengineering process. It was discovered that the original DFDs were out-of-date. In order to gain a consistent understanding of the system and to obtain a dependable set of documentation, the DFDs were analyzed and corrected. These analyses and corrections were based on information collected from the application system experts and the documentation, particularly the FSP.

It was concluded that any database redesign can have a significant impact on the manner in which application software accesses and processes data. Thus, any database redesign will force changes to the code that performs these functions. For example, in order to normalize the database to First Normal Form (1NF), it was necessary to eliminate repeating groups in each record definition. The original database definition was cluttered with repeating groups, making frequent use of the "OCCURS" clause in the DMS 1100 definition. Eliminating the redundancy forced changes to the application code that accessed the database and processed the data. In addition, redesign of some application code was necessary because of the change in navigation strategy. For example, one program sequentially processed database records by traversing the network structure of the database (i.e., get one record, process it, get next, and so on). This is perfectly suitable for hierarchical and network model databases, but not for relational databases in which record selection is based on the satisfaction of some criteria.

Once a relational database design that met our normalization requirements was developed, it was observed that the normalized design was not an optimum one. A more optimal design would have required a major redesign of the application code. It was realized that many design solutions were possible, and in order to adhere to the goals of the case study and time constraints, an optimal design was not necessary. Therefore, a database design solution that met the goal of 3NF was devised in which the impact on the application code was minimal.

The database conversion stressed to the members of the project team the importance of data reengineering and the essential coordination of the data side and process side of an application system during reengineering. When reengineering a system, it was felt that emphasis should be placed on data reengineering because it will drive the reengineering of the data processing code. This suggests that perpetuation of the data will be more useful than preservation of the original application process.

3.4 Findings

3.4.1 Process

It was determined that the complexity of the reengineering process increased in relation to the complexity of the programs. The most complex programs required the most manual effort. The program groups are listed below in order from the easiest (highly automated process overall) to the most difficult (much manual process required), with the breakdown of the required automated and manual effort provided as percentages. Note that the calculations below reflect the effort on the part of Booz, Allen only. The time dedicated by NIST and the IRS personnel is not included in these computations because the focus for the Government employees' time was on training and providing application system expertise to the contractor.

Program Group	Automated	Manual
Batch	96%	4%
Batch with SORT	90%	10%
Batch with DBMS access	88%	12%
Batch with SORT and DBMS access	82%	18%
Interactive	50%	50%

The majority of interactive programs were written in assembly language and therefore, this category required the most manual effort. Much time was spent determining the functionality of the assembler code and whether that functionality was still necessary in the target environment. It was calculated that overall 20% of the reengineering process was performed manually and 80% was performed automatically.

The following chart identifies the level of effort that was required to reengineer the system for each step in the methodology.

	Step	Percentage of Total Effort	Total Hours
Step 1	Baseline current system	19.76%	780
Step 2	Extract/analyze ...	43.26%	1,708
Step 3	Produce documentation	4.05%	160
Step 4	Generate new code	26.34%	1,040
Step 5	Execute and test new code	6.59%	260

Some of the CASE tools did not perform as advertised and required a greater than expected amount of manual intervention. On-hand, proprietary tools were modified to produce high-level design documentation. Some steps in the reengineering process seemed cumbersome and time-consuming. It was possible to automate some steps, but human effort was needed for analysis and tool operation. As a result of these difficulties, reengineering of the entire application system was not completed. Approximately 56% of the CSP system was reverse engineered to a design level and approximately 38% of the CSP system was reengineered (source code produced). Reengineering was completed on a representative sample of programs from each group. During an extended 18 week period, 24.7 staff months were expended. After accounting for the learning curve and problem resolutions, it was estimated that an additional 10 staff months would be necessary to complete the reengineering process.

Realizing the existence of other approaches for extending software lifetime, estimations were made in order to compare the effort required for reengineering with the effort for other approaches. The effort needed to convert the CSP system was calculated using the Office of Technical Assistance (OTA) Conversion Cost Model Version 4 and assuming the utilization of CASE tools. It was determined that approximately 30 staff months would be necessary to convert the CSP system. Possible reasons for this low number could be attributed to the large degree of automation of the conversion effort and that analysis would not be performed to gain higher-level design documentation.

The effort to redesign and redevelop the CSP system was calculated using the Constructive Cost Model (COCOMO) [BOEH81] for system development. It was assumed that CASE tools and 4GLs would be employed in this effort. Approximately 151.9 staff months would be necessary to redesign and redevelop a new system to meet the CSP requirements. The estimate of 34.7 staff months for reengineering compares favorably with these estimates.

The case study indicated that intimate knowledge of the original and target system platforms, the automated tools, and the implementation language is essential to carry out the reengineering process.

3.4.2 Metrics Analysis

In order to relate productivity and the quality achieved through the reengineering process, two analyses were performed on the accumulated measurements. Function point analysis was used to formulate indicators of productivity. Result metrics (metric counts before and after reengineering process) were analyzed to evaluate the degree to which the reengineering process affected maintainability and code flexibility. Caution must be used when evaluating the relevance of these calculations. Proper consolidation of function point measurements requires a significant sample size of similar reengineering projects that were all performed under the same conditions. Accordingly, this single project sample should not be regarded as sufficient for proper calibration. The metrics analysis is discussed in detail in Appendices A and B.

Through function point analysis on the reengineered programs, the following productivity measurements for the CSP system were derived:

- 164 function points per staff-year;
- 11.68 staff hours per function point;
- 1,516 executable statements per staff-year;
- 6,387 COBOL statements per staff-year.

As the sample size for this project is insufficient for drawing conclusions on the productivity of reengineering, the measurements above are presented for the purpose of information only.

Result metrics indicate that the reengineered programs are more complex than the original programs. This is evident in the increased number of logical NOT and GOTO statements. Also, the decision density is alarmingly high. Reasons for the increased complexity could be attributed to certain practices of the forward engineering tool. While the forward engineering tool has certainly eased the task of code generation, it has increased code complexity. It is important to note that increasing the code complexity may have a direct effect on the complexity of testing the code. The additional complexity could be justified if the code, hereafter, will be maintained at the design level and then forward engineered with the forward engineering tool. If maintenance returns to manual practices, then the task has been made more complex.

3.4.3 Reengineering Tools

The study indicated that the off-the-shelf tools used in this case study do not fully support the entire reengineering process. It was necessary to augment the off-the-shelf tools with modifications to available, proprietary tools. Most reverse engineering tools are analyzers, based on the recognition of certain structures, and are oriented towards a particular set of machines or environments (e.g., mainframe COBOL applications). However, this case study indicates the ability to maintain programs at a higher level of abstraction than at the source code level. Also, the code that was produced by the tool contained functions that can be time-consuming to code manually (i.e., record counting, file status checking, cleanup and housekeeping functions). This case study indicates that present CASE tools provide some efficiency and productivity gains, but further development of the reengineering technology is necessary. Human effort for design recovery and analysis is essential and can not be overlooked.

4. Conclusions and Recommendations

<u>4.1 General</u>

The following conclusions and recommendations are largely based on findings of this case study. Experience gained from other NIST projects involving reengineering also contributed to these recommendations. During the case study, only one application system was reengineered using a particular set of tools and a particular methodology. Accordingly, it would be incorrect to infer from this study absolute rules for when to reengineer and when to redesign. It must also be noted that the approach taken to reengineering was based on a certain set of goals for the study. Other reengineering approaches could be employed to achieve a different set of goals.

By comparing the efforts required to reengineer, convert, and redesign and develop the CSP system, it was concluded that software reengineering can be a cost-effective and viable solution to extending the lifetime of an application system. The cost-effectiveness and feasibility for reengineering a particular software system will be dependent on a number of variables that are specific to that system and the approach taken. These variables are: the goals for reengineering, condition of current application system and documentation, tool(s) support, and involvement of knowledgeable personnel.

NIST experience indicates that there is a spectrum of approaches that could be applied to extend the lifetime of a software application system as illustrated in figure 1. At one end of the spectrum is straight code conversion and at the opposite end is total redesign and development. In the middle of the spectrum are varying approaches of reengineering. These approaches are positioned between the endpoints in correspondence to the degree the original design and implementation is to be retained. The placement and separation between approaches on the spectrum is not clear cut. In each approach across the spectrum, higher-level abstractions of the system, in the form of analysis and design documentation, are derived. Reengineering approaches that are positioned on the spectrum include:

- reengineering with no change in design or functionality — design of original implementation is fully retained;

- reengineering with minimal change to design — only modifications necessary for target environment;

- reengineering with modifications to optimize the functionality and performance.

This case study focused on maintaining the functionality of the original system. Some redesign was necessary in order to convert the network database to a relational database structure that is normalized to 3NF. By positioning the approach for this case study around

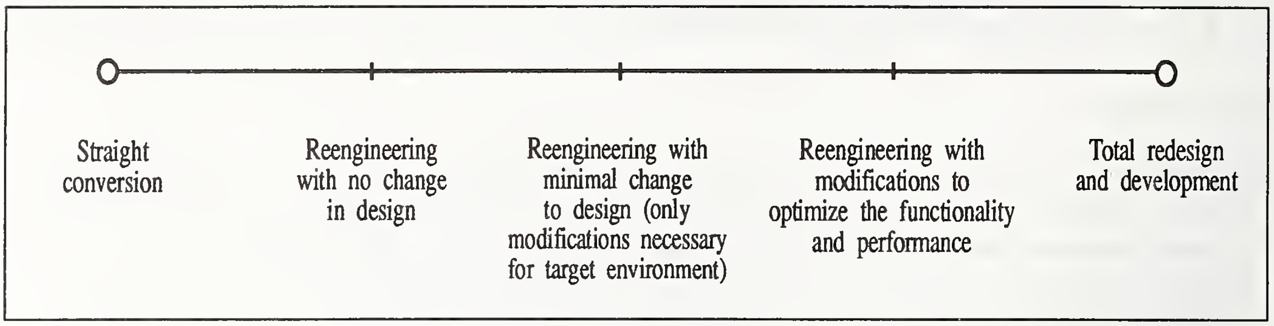
Straight conversion
Reengineering with no change in design
Reengineering with minimal change to design (only modifications necessary for target environment)
Reengineering with modifications to optimize the functionality and performance
Total redesign and development

Figure 1. Spectrum of Reengineering Approaches.

the center of the spectrum a set of guidelines for various reengineering approaches across the spectrum can be inferred.

The effectiveness of a reengineering approach is dependent on the variables of:

- corporate and system goals;
- condition of current application system and documentation;
- available resources (automated tool(s) support and personnel).

When determining if reengineering is an appropriate alternative for a particular application system and a given organization, the condition of these variables must be determined. The reason behind the success of this case study could be attributed to the strong correlation between the approach taken and these variables. This case study was supported by the following factors:

- well-defined focus and goals for this case study;
- original CSP system recently designed and implemented (1983);
- majority of original CSP code was well-structured (modularized with a minimum number of GOTOs);
- CSP was fairly well-documented — the primary documentation, the FSP, was up-to-date and complete;
- CASE tool support automated a majority of reengineering tasks;
- operations people who had excellent understanding of the original CSP system were involved throughout this project.

Reengineering can help an enterprise change its business functionality to meet its corporate goals. Through reengineering, changes to the way current business is done can be put into place. In order to be effective, upper management support is imperative. The degree of success of reengineering will be determined by the level of commitment made by the organization.

4.2 Corporate, System, and Reengineering Goals

■ When considering reengineering, one of the first tasks is to set the context for reengineering in terms of the corporate goals of the organization and the dependencies between application systems. For example, some Federal agencies (e.g., DoD) wish to implement some "corporate information management" policy in which the information used across the organization is effectively and efficiently controlled to eliminate inconsistencies, redundancies, and duplication of effort. The organization need not be global in scope, but could be at a local scope. Software reengineering is applicable in this context and would be most effective if directed at the information and its usage. A suitable approach for reengineering in the context of a corporate information management policy may be to reengineer the data and redesign the application processes. The various groups and application systems that use the same information should be identified along with their dependencies (i.e., what system creates the information, and what others display it or use it for further computations). Some redesign of the data and its usage may be necessary to implement a corporate information management policy. The processing of data in the original system is highly likely to change based on the redesign of data usage. Therefore the processing in the original system should not be a major influence on redesign or reengineering decisions. Organizations may have different corporate goals that will influence the effectiveness of software reengineering. These corporate goals and system dependencies must be considered to establish a context for reengineering.

Recommendation: **Establish the context for reengineering by considering the corporate goals of the organization and how reengineering could be applied to achieve the mission. Information dependencies between application systems must be identified.**

■ When examining the functional aspects of the current system, it is crucial to investigate the possibility of new solutions. Perhaps, using new technology, there is a better way to do business. The business may have been constrained, in the past, by technology. Indeed, the old manual methods, before computers, constrained the way business functions could be carried out. It was a mistake then to carry the manual methods into automation. Today it may be incorrect to carry current methods to newer architectures and approaches. If the current environment is driving the business approach, then it would be wise to reevaluate the technology currently being applied, the current business approach, and the business goals.

Recommendation: **Analyze system requirements from a functional viewpoint and consider new technology to improve current business practices.**

■ When determining a target computing architecture, conformance to government and industry standards should be considered. Because of the history of a computing environment, an organization may feel tied to a particular vendor for future purchases in order to achieve system compatibility. However, this argument is no longer accepted by the Government Accounting Office (GAO) in procurements for Federal agencies. Requiring equipment (hardware and software) to conform to appropriate standards (e.g., FIPS) can eliminate incompatibility problems, provide buyers with more flexibility when selecting equipment, and ease future migrations. Increasingly, Federal agencies are being mandated to procure equipment that conforms to FIPS, in particular the FIPS for POSIX, GOSIP, and SQL.

Recommendation: **When procuring equipment, require conformance to applicable standards (e.g., FIPS) to achieve flexibility and ease future migrations.**

■ It is important to have clearly defined motivations in order to determine a suitable reengineering approach. There are a number of factors to be considered. They include: current problems, the functional requirements and how they are currently met, new technology and how it can be exploited to improve satisfaction of the functional requirements, and an appropriate target environment. During the investigation of these factors, various motivations for reengineering will become clear. Because of the complexity of the reengineering process, the goals an organization expects to accomplish should be clearly stated before attempting to reengineer an application system. As discussed earlier, there are numerous reasons to consider reengineering such as:

- migrate to a new target environment;
- reduce maintenance costs;
- gain understanding of the current system's complexity;
- improve system performance;
- reduce software errors;
- recover information.

The selection of what is to be achieved by reengineering will assist in determining a suitable reengineering approach.

Recommendation: **Identify motivations and what is to be achieved by reengineering.**

4.3 Condition of Original System and Documentation

■ It is essential that the current condition of an application system be examined to determine if reengineering is practical, and if it is, how much redesign is required. Analysis of the dependencies between an application system and others that it may impact, in terms of information creation and usage, and functionality is needed. This calls for the system to undergo a thorough evaluation. In actuality, reverse engineering — gathering information about the system — is a large part of the system evaluation. The evaluation should be approached with the intent of discovering what is worth retaining for future use and what is not. Determining the extent to which modification is needed will narrow the field of choices of reengineering approaches. The system evaluation will determine if reengineering is practical and if it is, will identify the parts of the original design that should be retained, and necessary steps in the process. It is important to budget plenty of time for system analysis (reverse engineering) — it is a complex task and there are numerous aspects to consider.

Recommendation: Evaluate application system with the intent of discovering what is worth retaining for future use and what is not.

■ This case study stressed the importance of data and its usage in driving the application system. During evaluation, emphasis should be placed on the data side of the application rather than the process side. A functional perspective of data usage should be taken, and the system should be analyzed as to whether redesign of the data is needed. Redesign may be necessary because of poor original design, continual enhancements that have obscured the original design, inconsistent naming conventions, inconsistent data definitions, migration to a database of a different structure or the need to improve data management. Data redesign forces redesign of the processing code because of the change in definitions, access, and processing functionality. In some cases, data redesign will facilitate the removal of data processing code.

Considering the influence that the data side has on the application code, it would be practical to first evaluate the original data design and determine how much redesign is needed. Then an impact analysis should be performed to determine what programs are impacted and how. It may be that only the data access code will be effected, which is a minimal change. If the database is converted to a database of different structure, the functionality of how the application processes data may need to be modified. This evaluation strategy will narrow the choices of reengineering approaches.

Recommendation: Stress data design because it will force modifications to the process design.

■ Maintenance data and appropriate metrics should be analyzed for information concerning the system's history and performance. It should be determined whether the system conforms to software standards and the degree to which it conforms. To be reengineered the software should be in an extractable state — modularized software components with well-defined interfaces. Restructuring of the system may be necessary in order to prepare it for further reengineering. One possible reengineering approach is based directly on the application system's maintenance history. In this incremental strategy, the code that has the highest maintenance cost is reengineered first and work progresses on code segments that have decreasing maintenance costs.

Many organizations are currently operating application systems that are older than the CSP system and the code and documentation are in poor condition. Many older programs contain programming "tricks" for purposes of avoiding constraints of the environment or optimizing performance, such as saving memory or processing cycles. Reverse engineering from such code is dependent on an analyst's ability to recognize such sections of code.

It is possible that the results from the reverse engineering will indicate that the system is so error-prone and complex that perpetuation of the system is not practical. This conclusion eliminates the applicability of reengineering and narrows the choices to continue the use of the existing system or redesign and develop a new system. Another possibility is that the evaluation may indicate the need for some redesign. In those cases where a large percentage of redesign is required, it is better to scrap the system and redesign from scratch. A study of maintenance costs at IBM suggested that if 12% of a system has to be changed, then it is cheaper to redevelop [MART90a]; the study was too limited to support 12% as a generally applicable threshold.

Determining what parts of the system merit future use may also uncover redundant code segments. These segments do not have to be exactly the same, but may still be redundant even though they have slight differences. It may be possible to eliminate the redundancy by combining the redundant segments into one reusable part. Creating a reusable part from several redundant parts depends on whether the component can be extracted without significant effort. A redundant component that can be extracted as a distinct module with well-defined interfaces will be easiest to combine with other similar components. It may be valuable to create a library of reusable parts for use across application systems.

During system evaluation, asking the following types of questions will assist in determining if reengineering of an application system is suitable:

1. Does the original system's design and implementation merit reuse in a future system?

2. Are new technologies or methodologies exploitable that would improve satisfaction of the system requirements over the original application system?

3. Is the target environment vastly different from the original environment and if so, how?

4. Is redesign of the data necessary? If so, what parts on the process side will be impacted?

5. What parts of the system require redesign for operation in the target environment?

6. Is the system well-structured (modules with 1 entrance, 1 exit; no GOTOs)? If not, could it be improved by restructuring?

7. Is the system well-maintained?

8. Is the current performance of the system acceptable?

9. Is the documentation consistent and accurate with current system functionality?

Recommendation: Evaluate the code, documentation, maintenance history, and appropriate metrics to determine the current condition of the application system.

■ One motivation for reengineering is to gain a better understanding of the application system. The DFDs for the application system used in this study were out-of-date. In order to gain a more complete knowledge of the system and make the documentation consistent with the code, the documentation was analyzed, the application system experts were consulted and the DFDs were corrected. This was a long and difficult process because some information was recorded in other documentation forms while some was not recorded at all. This design recovery process served two purposes: to recover lost information and to assist the contractor personnel in gaining insight into the functionality of the application system.

Recommendation: While design recovery is difficult, time-consuming, and essentially a manual process, it is vital for recovering lost information and information transfer.

■ That information which is the most critical for understanding the system should be identified for preservation. All documentation forms should be analyzed to determine what information is stored and how important it is to understanding the system. For example, the application system experts relied most heavily on the FSP for functional information of the system — hence it represented the most up-to-date documentation. The focus here is not on documentation form, but on the content. It is important not to be strongly tied to a certain set of documentation forms. It is useful to have standards for documentation, but it is critical to be open for better ways of representing and maintaining information. Current documentation practices are quite limited in the types of information that are maintained. While it may be necessary to restrict information in a documentation form to ensure consistency and completeness, it is important to analyze other perspectives of the system to gain as complete an understanding of the system as possible. Currently formal description techniques, such as SDL [CCITTRB], SPEC [BERZ90], Estelle [ISO9074], LOTOS [ISO8807], are being utilized for design and documentation of systems. Some of these techniques may be more appropriate for documentation and maintenance than the current practices.

In addition, current document practices should be analyzed as to how useful each is for various purposes. For example, DFDs are useful in the software design phase for identifying processes and the data needed by each process. However, DFDs can quickly become overly complex. This was evident in this study — the highest level DFD looked like the physical layout of a micro-chip, despite consolidation of the data streams and process bubbles. This complexity might well be the reason why reliance during maintenance is placed on the FSPs, rather than the DFDs.

Recommendation: **Identify critical system information. Do not be tightly tied to a certain set of documentation forms; focus on information content and usage.**

4.4 Resources

It became apparent during this study that resources (automated tools support and personnel) are a critical factor in the successful completion of a reengineering project. The selection of a software engineering methodology and the associated CASE tools is of paramount importance because of their impact on the future operation and maintenance of the reengineered system. The reengineering process can add additional overhead to the system (i.e., more documentation, increased complexity of the generated code). This additional overhead is justifiable if the system will be maintained at the design level and then forward engineered with the methodology used in the reengineering process. If maintenance returns to manual practices or employs different tools that do not use the information recorded in the repository, then much of the gains from the reengineering effort will be lost.

4.4.1 Automated Tool Support

■ This case study indicates that the support for software reengineering from currently available off-the-shelf CASE tools is far from the ideal situation of totally automated reengineering — building high-level design and analysis information based on low-level descriptions (code) and then forward engineering to an environment of choice. Although present CASE tools do provide much efficiency and productivity gains, the technology needs further development to provide a complete set of needed functions. CASE tools do not always perform as advertised, may require manual intervention, and some steps may be cumbersome and time-consuming. It was necessary to augment the CASE tools used in this study with on-hand proprietary tools. In many cases, modifications were made to the proprietary tools to resolve problems encountered or lack of support from off-the-shelf tools.

Recommendation: **Provisions in terms of personnel and effort must be made to compensate for the lack of full support of the reengineering process by currently available off-the-shelf tools.**

■ Most currently available CASE tools are directed at one particular aspect of software engineering (and reengineering) and are targeted for a certain environment. Application systems within an organization may differ drastically in terms of environment and methodology. Because of the differences across applications and the targeting of tools, it should not be assumed that a single toolset will apply uniformly well across all application systems. It may be suitable to utilize a number of tools, each being used for its particular strength in the reengineering process and supported environment.

Recommendation: **Considering the focus of most CASE tools for a particular computing environment, one set of CASE tools should not be depended on for uniform applicability to all needs across an organization.**

■ It is essential that the hardware that supports the reengineering process have adequate storage capacity and processor speed. With the methodology and tools used in this study, large files were generated during the reengineering process. If sufficient processor speed is not used, the reengineering process could be inhibited.

Recommendation: **Adequate storage capacity and processor speed in equipment supporting the reengineering tools are essential to facilitate the reengineering process.**

■ Before reengineering a system, decision makers should consider if the tools chosen to handle this procedure follows any particular methodology. The current methodologies utilized by an organization may change drastically once an application system is reengineered. These changes may cause frustration for individuals working with the system.

Recommendation: **Consider CASE reengineering tools that provide methodologies which are compatible to the requirements of the particular enterprise.**

■ When reengineering an application, it may be necessary to use multiple tools from different vendors. This may cause problems with the interchange and integration of data and data models across different tools. A data model provides a method for representing the data structures used in a software engineering toolset or repository. Different tools may not support the same data models and methodologies resulting in the need for data model integration.

There are additional CASE tool features that are worthy of consideration, such as export/import and appropriate metrics analysis. This, of course, is dependent on the functional requirements of the CASE tools. For example, in order to accomplish data model integration when using different software engineering tools, the user organization must be able to recognize the similarities and differences between the different data models in use. One means of achieving data model integration is by identifying the differences between the data models and building a target data model that will be sufficiently robust so as to be capable of capturing both data models. When toolsets or tool dictionaries have fixed, non-extensible data models, then data model integration becomes difficult or even impossible to accomplish. An export/import interchange facility can provide some support for interchange of data models and data. Also, it is beneficial to utilize CASE tools that support data collection and appropriate metrics analysis. Having an automated means for data collection is superior to manual methods because of savings in time and labor. Also, automated methods ensure that the data are collected and measurements are made in a consistent manner.

Recommendation: **Additional features that merit consideration include a data interchange facility and appropriate metric analysis utility.**

4.4.2 Personnel

■ Reengineering requires a highly trained staff that has experience in the current and target system, the automated tools, and the specific programming languages. It is not necessary that all the reengineering team members have all of these skills, but these skills must be present across the team. If it is desired to automate the reengineering process as much as possible, team members who are able to write additional software to bridge the gaps between the CASE tools and/or provide special support for the tools may be required.

Recommendation: **Reengineering requires a highly trained staff that has experience in the current and target system, the automated tools, and the specific programming languages.**

■ Human knowledge and understanding of the application system to be reengineered is extremely important. Without the involvement of the application system experts, this study could not have been completed. While the documentation was helpful, some sections were out-of-date and the application system was quite complex. With their knowledge and experience, the human experts were able to supply complete information of the system that could not have been gained from the documentation alone. However, shifting application system experts from maintenance to reengineering implies that considerable staff hours may need to be diverted from operational work.

Recommendation: **It is critical that the application system experts be involved throughout the reengineering process. They are essential for design recovery.**

5. Final Remarks

This document discusses software reengineering (and related terminology) and how reengineering can be used to extend the lifetime of existing software. The use of CASE tools for the support of reengineering and various tool considerations have been examined. Through the completion of a case study directed at evaluating the feasibility and cost-effectiveness of software reengineering, some preliminary results have been determined. Software reengineering can be a cost-effective and viable solution for extending the lifetime of an application system. The degree to which it is cost-effective is dependent on the goals for reengineering, the condition of the original application system and documentation, available automated tool support, and the involved personnel. These variables must be thoroughly analyzed before selecting a reengineering approach. This approach determination analysis is essential and must not be overlooked. Factors to be considered when determining a reengineering approach were addressed. These factors range from corporate goals to the condition of the original system and resource support.

6. References

[ARAN85] Arango, G., Baxter, I., Freeman, P., and Pidgeon, C., "Maintenance and Porting of Software by Design Recovery," Proceedings from Conference on Software Maintenance 1985.

[ARNO90] Arnold, R., Notes from Seminar on Software Reengineering.

[BACH88] Bachman, C., "A CASE for Reverse Engineering," Datamation, July 1, 1988.

[BASI90] Basili, V., "Viewing Maintenance as Reuse-Oriented Software Development," IEEE Software, January 1990.

[BERZ90] Berzins, V., et al, "An Introduction to the Specification Language Spec," IEEE Software, March 1990.

[BIGG89] Biggerstaff, T., "Design Recovery for Maintenance and Reuse," COMPUTER, July 1989.

[BOEH81] Boehm, B.W., Software Engineering Economics, Prentice-Hall, Inc., 1981.

[BOOZ91] Booz, Allen & Hamilton, Reverse Engineering Evaluation Process Report, January 15, 1991. This is an internal, restricted report.

[CCITTRB] CCITT Red Book Volume VI-Fascicle V1.10, "Functional Specification and Description Language (SDL)."

[CHIK90] Chikofsky, E., and Cross, J., "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, January 1990.

[CHOI90] Choi, S., and Scacchi, W., "Extracting and Restructuring the Design of Large Systems," IEEE Software, January 1990.

[DIEH89] Diehl S., et al, "Making a Case for CASE," BYTE, December 1989.

[FAIR85] Fairley, R., Software Engineering Concepts, McGraw-Hill Book Co., 1985.

[GANE90] Gane, C., Computer-aided Software Engineering the methodologies, the products, and the future, Prentice-Hall, Inc., 1990.

[HAUG91] Haugh, J., "A Survey of Technology Related to Software Reengineering," Proceedings from Systems Reengineering Workshop, Naval Surface Warfare Center, Silver Spring, Maryland, March 25-27, 1991

[IFPUG90] Sprouls, J., (ed.), IFPUG Function Point Counting Practices Manual, Release 3.0, 1990.

[ISO8807] International Organization for Standardization, "Information processing systems — Open systems interconnection –– LOTOS — A Formal description technique based on the temporal ordering of observational behavior," 1988.

[ISO9074] International Organization for Standardization, "Information processing systems — Open Systems Interconnection — Estelle — A formal description technique based on an extended state transition model," 1989.

[JONE88] Jones, Capers, "A Short History of Function Points and Feature Points," Software Productivity Research, Inc., Version 2.0, Feb. 20, 1988.

[KOZA91] Kozacynski, W., "A Suzuki Class in Software Reengineering," IEEE Software January, 1991.

[MART90a] Martin, J., "The Beauty of Re-Engineering: Continual Enhancements," PC Week, April 30, 1990.

[MART90b] Martin, J., "Restructuring Code Is a Sound Investment in the Future," PC Week, May 7, 1990.

[RICK89] Ricketts, J.A., DelMonaco, J.C., Weeks, M.W., "Data Reengineering for Application Systems," Proceedings from Conference on Software Maintenance, 1988.

[ROSE89] Rosen, Bruce K., and Law, Margaret H., "Information Resource Dictionary System (IRDS) and Modeling Tools," Proceedings of 3Rs of Software Automation, Re-engineering, Reusability, Repositories, An Extended Intelligence, Inc. Conference and Tool Exhibition, 1989.

[RUHL91] Ruhl, M., IRS Software Reengineering Report and Strategy Plan, January 30, 1991. This is an internal, restricted report.

[SHAR91] Sharon, D. "CASE Standards: Is Anyone Listening?," CASE Trends, March/April 1991.

[SNEE87] Sneed, H., and Jandrasics, G., "Software Recycling," Proceedings from Conference on Software Maintenance, 1987.

[WEIN91] Weinman, E., "The Promise of Software Reengineering," InformationWeek, April 22, 1991.

Appendix A: Function Point Analysis

Function point analysis was chosen to measure the productivity achieved in the reengineering process, as well as to measure the degree of functionality of the original CSP system. Function point analysis has, in many cases, been proven to be superior to conventional metrics based on lines of code (LOC) count. Such conventional metrics would have posed several problems in this reengineering study since:

- CSP consisted of COBOL, assembly language code, and DBMS commands, making a reduction to normalized LOC difficult;

- LOC is usually a meaningless measure for systems making significant use of DBMS;

- there are no standard scope of effort guidelines [BOOZ91].

Function point analysis is based on measurements of inputs, outputs, inquiries, master files, and interfaces, each of which is appropriately weighted. The impacts of possible influential factors are analyzed to determine the level of system complexity. This provides a dimensionless number as an indicator of functionality. The International Function Point Users Group (IFPUG) publishes counting rules and guidelines to ensure consistent definitions and counting methodology [IFPUG90]. As the sample size for this project is insufficient for drawing conclusions on the productivity of reengineering, the measurements below are presented for the purpose of information only.

Function point analysis was first performed on the CSP system before reengineering in order to gain some idea of its condition. Function point analysis was also performed on the 14 programs that were reengineered.

A final function point count of 1,192 for the entire CSP system before reengineering was derived. Considering the size of the reengineered program inventory (approximately 50,000 total COBOL source statements), the function point total is high. Most published reports associate 100 COBOL source statements per function point. Following this estimate, less than half the function points found would be expected for this application system. The main conclusion that can be drawn from this is that it is incorrect to expect a high correlation between function points and lines of code count. Secondly, the analysis of the original inventory indicated that the original code was well-structured with a high degree of functionality. It is not improbable to derive a high function point count to code ratio for well-structured, highly functional programs.

Reengineering was completed on 14 programs. The measurements for these programs before the reengineering process are as follows:

- 13,131 COBOL source statements,

- 3,116 executable statements,
- 338 function points.

The following measurements were obtained for the sample programs after reengineering:

- 21,480 COBOL source statements,
- 4,062 executable statements.

The reengineering process required a total of 3,948 staff hours. Assuming 1,920 staff-hours per work year, 2.056 staff years were expended. Because the goal was to reengineer without any changes to the functionality and the high level of automation of the reengineering process, the productivity measurements were based on the measurements before reengineering. The productivity measurements are listed below:

- 164 function points per staff-year,
- 11.68 staff hours per function point,
- 1,516 executable statements per staff-year,
- 6,387 COBOL statements per staff-year.

Because of the significant use of DBMS in this application system, the statements per staff-year measurements should not be considered highly meaningful. Rather, the 164 function points per staff-year is the key parameter [BOOZ91].

Appendix B: Result Metrics Analysis

The result analysis focused on how the reengineering process affected maintainability and code flexibility. The classifications chosen for the totals reflect this focus. The measurements were made using an automated tool which collects and categorizes statement counts.

The following counts were used as a basis for comparison:

- ELOC: Executable lines of code;
- CLOC: ELOC divided by 100;
- Size: CLOC**2 (Note: the choice of power 2 was arbitrary; the intent was to penalize programs exceeding 100 ELOC by a significant margin);
- Decision count: Count of all decision statements (if, do-while, do-until, etc.);
- Decision density: Number of decision statements per CLOC;
- Function count: Number of COBOL functional statements (call, perform, compute, sort, merge, etc.);
- Number of COBOL I/O statements;
- Entry/Exit Ratio (EER): Number of ENTRY statements per program exit (EXIT, GOBACK, and STOP RUN statements).

Computed values were normalized to a base of 100 executable lines of code (CLOC). This was done to allow meaningful analysis between before and after measurements and so that structured code would not be penalized — structured code typically results in more lines of code than equivalent unstructured code, although the number of executable lines of code is no higher. Additionally, these derived metrics were divided by the number of reengineered programs in order to provide an averaged measurement.

Decision counts are closely related to complexity and testability. A decision density (decisions per CLOC) of 10 or less is desirable. When decision density exceeds 20-25, it is likely that maintenance problems will be experienced. The number of functional statements in a program is an indicator of a module's function strength. A function density of over 20% is a good indicator of a highly functional, and therefore easily maintainable, module. Conversely, a GOTO count of over 10 will cause maintenance problems — a count of 2-3 is desirable. Additionally, a large number of NOT logical statements will cause problems since NOT statements test what is not and give no indication of what is. Well-modularized programs will contain balanced counts of ENTRY/EXIT statements. A comment density of 10% or more is desirable for COBOL code [BOOZ91].

Table 1 displays the key result metrics counts before and after the reengineering process.

Table 1. Key Result Metrics Counts

Metric	Before Reengineering	After Reengineering
Total Executable Lines of Code	3116.00	4062.00
Number of Programs	14.00	14.00
CLOC	2.23	2.90
Total Decision Count	710.00	1376.00
Decision Density	22.74	33.89
Total Function Count (COBOL)	537.00	1407.00
Total I/O Count	384.00	335.00
Entry/Exit Ratio (EER)	0.50	0.66
Size	4.95	8.41
GOTO Density	0.25	6.67
NOT Clauses	159.00	466.00
Functions/Programs	38.36	100.50
Function Density	17.20	34.66
Total Number of Files	46.00	57.00
Total Number of Calls	42.00	8.00
Comments Density	118.87	318.04

The metrics before reengineering present a mixed picture. The size is relatively small (1 is ideal). The GOTO density is very small, but there are a large number of NOT clauses. Function density is high, which is good, but decision density is also high. A high decision den sity is indicative of complex code. The relatively high I/O count also indicates a high level of complexity. These sample programs could be characterized as well-written, highly structured and modular, overly documented, and somewhat complex.

The result metrics indicate that the reengineered programs are more complex then the original programs. This is evident from the increase in the number of logical NOT and GOTO statem ents. Also, the decision density is alarmingly high. Size has increased while the I/O count has decreased slightly. Function density is nearly doubled (mostly due to PERFORMs) [BOOZ91].

Reasons for this added complexity could be attributed to certain practices of the forward engineering tool. While the forward engineering tool has certainly eased the task of code generation, it has increased code complexity. It is important to note that increasing the code complexity may have a direct effect on the complexity of testing the code. The additional complexity could be justified if the code, hereafter, will be maintained at the design level and then forward engineered with the forward engineering tool. If maintenance returns to manual practices, then the task of maintaining the program has been complicated.

Appendix C: Glossary

CASE Computer-Aided Software Engineering
The creation of software systems using a well-defined design technique and development methodology, supported by computer-based automation tools.

CASE tool
A software program that provides partial or total automation of a single function within the software life cycle.

Data reengineering
"a system level process that purifies data definitions and values." This process "establishes meaningful, nonredundant data definitions and valid, consistent data values" [RICK89].

Design recovery
"a subset of reverse engineering in which domain knowledge, external information, and deduction of fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself" [CHIK90].

Forward engineering
"the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system" [CHIK90].

GOSIP Government Open System Interconnection Profile
This Federal Information Processing Standard is intended to simplify and ease the process of assimilating OSI technology in the Federal agencies. GOSIP defines and describes a common set of data communication protocols which enable systems developed by different vendors and enable the users of different applications on these systems to exchange information.

IRDS Information Resource Dictionary System
The IRDS Standard is a set of software specifications for a standard data dictionary system. It "establishes the requirements for a software tool that can be used to describe, document, protect, control, and enhance the use of an organization's information resources" [ROSE89].

POSIX Portable Operating System Interface
This is an ongoing effort within IEEE to standardize an operating system interface for the purpose of development of portable software. A number of government agencies plan to achieve portability by conforming to this Federal Information Processing Standard (FIPS).

Redocumentation

"the creation or revision of a semantically equivalent representation within the same relative abstraction level" [CHIK90].

Restructuring

"the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (function and semantics)" [CHIK90].

Reverse engineering

"the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [CHIK90].

Software reengineering

"the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form" [CHIK90]. Reengineering is also known as renovation and reclamation.

NIST-114A
(REV. 3-90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER	NIST/SP-500/193
2. PERFORMING ORGANIZATION REPORT NUMBER	
3. PUBLICATION DATE	September 1991

4. TITLE AND SUBTITLE

Software Reengineering: A Case Study and Lessons Learned

5. AUTHOR(S)

Mary K. Ruhl and Mary T. Gunn

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

Same as item #6

10. SUPPLEMENTARY NOTES

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

This report is aimed at managers and technical personnel (both Federal Government and industry) who need to understand:

- the concepts and issues of software reengineering,
- the use of Computer Aided Software Engineering (CASE) tools in the reengineering process,
- and the application of this technology to organizational problems.

Software reengineering involves the use of existing software and documentation to specify requirements, design, documentation, and to produce software for a target platform. CASE tools are expected to play an important role in automating parts of the reengineering process.

In this report software reengineering and other related terms are defined and possible benefits that relate to this technology are described. The use of CASE tools for reengineering are examined. A case study that examines the feasibility and cost-effectiveness of software reengineering is described. Study results are addressed along with recommendations for organizations that are considering the use of reengineering.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

CASE (Computer-Aided Software Engineering) tools, design recovery, reengineering strategies, reverse engineering, software reengineering.

13. AVAILABILITY

X	UNLIMITED
	FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
X	ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
X	ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES

39

15. PRICE

ELECTRONIC FORM

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SYSTEMS TECHNOLOGY

Superintendent of Documents
Government Printing Office
Washington, DC 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Institute of Standards and Technology Special Publication 500-.

Name ____

Company ____

Address ____

City ______________ State ______________ Zip Code ______________

(Notification key N-503)