NSU NOVA SOUTHEASTERN UNIVERSITY NSU Libraries

Nova Southeastern University
NSUWorks

CEC Theses and Dissertations | College of Engineering and Computing

1996

Design Information Recovery from Legacy System COBOL Source Code: Research on a Reverse Engineering Methodology

Robert Lee Miller
Nova Southeastern University, millerrl@att.net

This document is a product of extensive research conducted at the Nova Southeastern University College of Engineering and Computing. For more information on research and degree programs at the NSU College of Engineering and Computing, please click here.

Follow this and additional works at: http://nsuworks.nova.edu/gscis_etd

Part of the Computer Sciences Commons

Share Feedback About This Item

NSUWorks Citation

Robert Lee Miller. 1996. *Design Information Recovery from Legacy System COBOL Source Code: Research on a Reverse Engineering Methodology.* Doctoral dissertation. Nova Southeastern University. Retrieved from NSUWorks, Graduate School of Computer and Information Sciences. (727)
http://nsuworks.nova.edu/gscis_etd/727.

This Dissertation is brought to you by the College of Engineering and Computing at NSUWorks. It has been accepted for inclusion in CEC Theses and Dissertations by an authorized administrator of NSUWorks. For more information, please contact nsuworks@nova.edu.

Design Information Recovery from Legacy System COBOL Source Code: Research on a Reverse Engineering Methodology

by

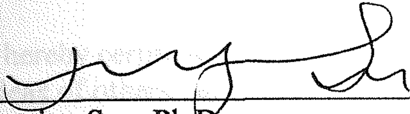
Robert Lee Miller

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

School of Computer and Information Sciences
Nova Southeastern University

1996

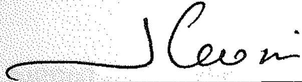
We hereby certify that this dissertation, submitted by Robert L. Miller, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.



10/31/96

Junping Sun, Ph.D.
Chairperson of Dissertation Committee

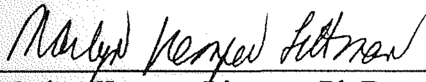
Date



11/18/96

Jacques Levin, Ph.D.
Dissertation Committee Member

Date



11/20/96

Marlyn Kemper Littman, Ph.D.
Dissertation Committee Member

Date

Approved:

12/11/96

Edward Lieblein, Ph.D.
Dean, School of Computer and Information Sciences

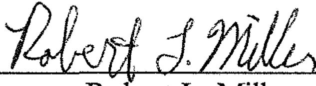
Date

School of Compeer and Information Sciences
Nova Southeastern University

1996

Certification Statement

I hereby certify that this dissertation constitutes my own product and that the words or ideas of others, where used, are properly credited according to accepted standards for professional publications.

Signed 
Robert L. Miller

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Design Information Recovery from Legacy System COBOL Source Code: Research on a Reverse Engineering Methodology

by
Robert L. Miller

October 1996

Much of the software in the world today was developed from the mid-1960s to the mid-1970s. This legacy software deteriorates as it is modified to satisfy new organizational requirements. Currently, legacy system maintenance requires more time than new system development.

Eventually, legacy systems must be replaced. Identifying their functionality is a critical part of the replacement effort. Recovering functions from source code is difficult because the domain knowledge used to develop the system is not routinely retained. The source code is frequently the only reliable source of functional information.

This dissertation describes functional process information recovery from COBOL source code in the military logistics system domain. The methodology was developed as an information processing application. Conceptual and logical models to convert source code to functional design information were created to define the process. A supporting data structure was also developed.

The process reverse engineering methodology was manually applied to a test case to demonstrate feasibility, practicality, and usefulness. Metrics for predicting the time required were developed and analyzed based on the results of the test case.

The methodology was found to be effective in recovering functional process information from source code. A prototype program information database was developed and implemented to aid in data collection and manipulation; it also supported the process of preparing program structure models.

Recommendations for further research include applying the methodology to a larger test case to validate findings and extending it to include a comparable data reverse engineering procedure.

ACKNOWLEDGMENTS

Completing a doctoral dissertation is a long and arduous process that requires extraordinary dedication, commitment, and perseverance. Throughout this effort, support was provided by a group of people to whom I owe a deep debt of gratitude. For without their support, this research project would have never been completed.

I am indebted to my dissertation advisor, Dr. Junping Sun, who offered critical encouragement, advice, and criticism. I also wish to thank Dr. Jacques Levin and Dr. Marlyn Kemper Littman, dissertation committee members. They guided me in the transformation of a fuzzy concept into a worthwhile research project.

I am grateful to the staff and faculty of the Nova Southeastern University School of Computer and Information Science. They are, without exception, dedicated to assisting the doctoral student.

A co-worker, JoAnn (jody) Morley listened patiently and provided valuable ideas and suggestions during preliminary applications of the concepts upon which this research is based. Thanks for your confidence, jody.

Many thanks to Heather Pierce, also a co-worker, who read the early drafts and made sure they were written in English. Her help was invaluable in keeping my writing concise and to the point.

Finally, my deepest gratitude and appreciation to my wife, Fumiko, for her understanding, tolerance, and support.

R.L.M.

Table of Contents

Abstract iii

Acknowledgments iv

Table of Contents v

List of Tables xiii

List of Figures xv

Chapter

I. Introduction 1

Statement of the Problem 1

Barriers and Issues 2

Barriers to Reverse Engineering 2

Issues in Reverse Engineering 3

Importance of the Topic 4

Significance of the Research 7

Definition of Terms 8

Brief History of Reverse Engineering 9

Objectives of the Research 10

Scope of the Research 11

Research Questions Investigated 12

Research Methodology 13

Phase I: Approach Selection 13

Phase II: Methodology Development 13

Phase III: Case Problem Selection 14

Phase IV: Methodology Application 14

Phase V: Methodology Assessment 15

Limitations and Delimitations of the Research 15

Limitations of the Research 15

Delimitations of the Research 16

Contributions of the Research 17

Theoretical Contribution 17

Managerial Contribution 17

Criteria for Success 19

Documented Reverse Engineering Methodology 20

Validated Methodology 20

Program Information Database Conceptual and Logical Data Models 21

Work Load Estimation Metrics 21

Summary 21

II. Review of the Literature 23

Introduction 23

Programming Languages 25

Programming Language Syntax 26

Programming Language Semantics 27

Programming Language Components 28

Job Control Language (JCL) 29

The COBOL Language 30

COBOL History 31

COBOL Structure 32

Nature of COBOL 41

COBOL Disadvantages 43

COBOL Advantages 49

Program Understanding 49

The Meaning of Program Understanding 52

Software Psychology 53

Factors Affecting Program Understanding 55

Domains and Program Understanding 59

Approaches to Program Understanding 61

The Need for Reverse Engineering 63

Legacy Systems 64

Software Aging 66

Business Changes 68

Business Process Reengineering 68

Client/Server Technology 69

Object-Oriented Technology 70

Software Maintenance 70

Reverse Engineering Economics 71

Reverse Engineering 74

Reverse Engineering Objectives 82

The Basis for Reverse Engineering 83

Reverse Engineering Problems 84

Design Recovery/Inverse Engineering 90

Existing Reverse Engineering Procedures 91

Software Physical Structure 97

Knowledge-based Program Understanding Tools 98

Transformation Tools 99

Program Plans 99

Data Flow Diagrams 100

Functional Abstraction Tools 100

Computer Assisted Reverse Engineering (CARE) Tools 101

Summary 104

III. Methodology 109

Research Methods Employed 110

Specific Procedures Employed 111

Description of Phase 1 - Reverse Engineering Approach Selection 111

Description of Phase 2 - Reverse Engineering Methodology Development 113

Description of Phase 3 - Case Study Subject Selection 114

Description of Phase 4 - Reverse Engineering Methodology Application 114

Description of Phase 5 - Methodology Assessment 115

Execute the Reverse Engineering Synthesis Plan 115

Problem Definition 116

The Operational Environment 116

Operational Problems 117

COBOL Program Environment 118

A Forward Engineering Model 120

Essential Points in the Forward Engineering Model 121

Information Loss in Forward Engineering 128

A Model of the Reverse Engineering Process 131

Differences in Forward and Reverse Engineering 131

Problems Associated with Reverse Engineering 134

A Formal Method of Reverse Engineering - Clean-Specify-Simplify 140

A Structural Approach to Reverse Engineering - Program Schematics 143

A Program Understanding Approach to Reverse Engineering - DESIRE 151

A Data-Oriented Reverse Engineering Technique - Component Extraction 155

A Data Repository Approach to Reverse Engineering - System Description Database 160

Reverse Engineering Approach for Air Force Logistics Systems 164

A Model On-line Program 166

A Model Batch Program 169

A Model Fourth Generation Language (4GL) Program 171

Reverse Engineering Process Output Products 173

Developing the Reverse Engineering Methodology 178

Purpose 178

Scope 178

Strategy 179

Goals 183

The Conceptual Process Model 184

The Conceptual Data Model 189

The Definitional Model 196

The Logical Model 199

The Logical Process Model 199

Logical Modeling Technique 199

Logical Model Services 201

The Logical Data Model 250

Formats for Presenting Results 252

Projected Outcomes 252

Resource Requirements 253

Reliability and Validity 253

Summary 254

Chapter IV. Results 255

Data Analysis 255

Implementing the Program Information Database 256

Selecting Case Study Components 257

Applying the Methodology 265

Findings 270

Documentation 270

Program Source Code 273

Job Control Language 274

Methodology Assessment 275

Comparison of User and Derived Function Model 277

Metrics 278

Source Lines of Code 284

Procedure Division Lines of Code 288

Number of Procedure Division Paragraphs 291

Complexity Index 294

Metrics Analysis Summary 298

Methodology Changes 300

Summary of Results 302

Chapter V. Conclusions

Conclusions 304

Implications 307

Recommendations 308

Summary 309

Appendix

A. Glossary 315

B. Reverse Engineering Methodology Conceptual Model Data Flow Diagrams and Process Descriptions 322

C. Reverse Engineering Methodology Logical Data Model Table Descriptions 404

D. Permission to Use Source Code 417

E. Biographical Sketch of Student 419

References 421

List of Tables

Table

1. ANSI-74 COBOL Consists of a Nucleus and 11 Modules 33

2. COBOL Program Field Assignments 34

3. General Structural Elements of COBOL Programs 37

4. Reverse Engineering Tools and Methodologies 92

5. Nominal Reverse Engineering Tools Available Commercially 102

6. The Nine Subsystems Vary in Size and Programming Language 119

7. List of Functional Units 145

8. Linear Circuits 149

9. Application 1 - LC Components 150

10. Conceptual Data Model Entity List 192

11. Conceptual Data Model Relationship List 194

12. Skeletal Table List 197

13. Batch Programs 260

14. IDEAL (On-line) Programs 260

15. CICS COBOL (On-line) Programs 261

16. Time Required for Preliminary Program Review 267

17. Time Required for Job Control Language Review 268

18. Time Required for Detailed Program Structure Analysis 269

19. Time Required for Function Analysis and Process Assignment 271

20. Example of COBOL Data Names and Equivalent Full-Text Data Names 275

21. Paragraphs Extracted from Programs and Used in the Domain Model 278

22. Number of Program Paragraphs Allocated to Functions 279

23. Summary of Program Review and Analysis Time 281

24. ANOVA for the SLOC Linear Regression Analysis 286

25. ANOVA for the PDLOC Linear Regression Analysis 289

26. ANOVA for the NOPARA Linear Regression Analysis 292

27. ANOVA for the CI Linear Regression Analysis 296

28. Summary of Regressions Analysis Results 298

29. Comparison of Initial, Actual, and Computed Reverse Engineering Time 299

List of Figures

Figure

1. Reverse engineering and related processes are transformations between or within abstraction levels 10
2. Sample COBOL coding sheet 35
3. Structural elements of COBOL programs 36
4. COBOL Environment Division structure 39
5. Sample COBOL program 44
6. System forward engineering and reverse engineering 60
7. Decision matrix: what to do with an old system 73
8. Software option strategy matrix 73
9. Virtual overlapping between code and program design 87
10. Forward engineering process model A0 diagram 122
11. Process A1 decomposition 123
12. Process A2 decomposition 124
13. Process A3 decomposition 125
14. Process A23 decomposition 126
15. Software system development knowledge transfer 127
16. Reverse engineering A0 process model diagram 132
17. Network of non-subroutine units 147
18. Trunk of the tree LC 148
19. Three kinds of recovered components 157

20. Model CICS on-line program 167

21. Model COBOL batch program 170

22. Model 4GL program 172

23. External interfaces of a system component 182

24. Context diagram 186

25. Level 0 diagram 187

26. Conceptual data model 190

27. Sample program implementation model 238

28. Logical Data Model 251

29. SLOC scatterplot 282

30. PDLOC scatterplot 283

31. NOPARA scatterplot 283

32. CI scatterplot 284

33. SLOC regression line 285

34. Residuals plot SLOC 287

35. Regression line PDLOC 288

36. Residuals plot PDLOC 290

37. NOPARA regression line 292

38. NOPARA residuals plot 294

39. CI regression line 295

40. Residuals plot CI 297

Chapter I

Introduction

Computing, also termed data processing, information systems, or computer science, is a relatively new human activity. Depending on when the beginning of the era of widespread electronic computer use is established, computing is 30 to 40 years old. Thus, there is no significant history of computing as is found with other human endeavors. For example, mathematics and chemistry have histories that extend hundreds or even thousands of years. Modern science in general is based on well established axioms, principles, hypotheses, and theories. This is not the case with data processing, nor in particular with the software development process. Although research continues, there is still much not known, nor well understood, about computer software and the way it is developed. Even less is known about extracting system design information from existing system software--a process called *reverse engineering*.

Statement of the Problem

In the world today there are billions of lines of COBOL program source code representing legacy systems that are 20 or more years old. Maintaining these systems is time-consuming and extremely expensive because the code is difficult to understand. The

system documentation is incomplete, incorrect, or outdated. The source code is often the only reliable source of information regarding the functions performed by the system and the business rules under which the functions are carried out. Replacing legacy systems is complex because it is difficult to extract design information from COBOL source code. By using a formal reverse engineering methodology, the systems analyst or maintenance programmer can extract design information from legacy system source code quickly and with accurate results to support systems replacement or systems maintenance.

Barriers and Issues

Barriers to Reverse Engineering

A major barrier to reverse engineering is methodologists' tendency to ignore software maintenance and devote most of their attention to developing new systems (Brittain, 1991). Another barrier to reverse engineering is the lack of automated tools to support the process. Kerr and McGovern (1991) predicted reverse engineering would reach maturity in 1995 with a standard repository design and full-function reverse engineering tools. It is significant that this prediction appeared in a trade magazine; such bold predictions are seldom, if ever, seen in formal technical journals. Five years after this prediction not only is there no standard repository design, there is still no full-function reverse engineering tool.

Desmond (1992) observed that the promise of automated tools was that they would make it possible to abstract the logic of an old application and reapply that abstraction in a new

application. He suggested that software professionals, however, have learned how difficult it is to extract business rules from COBOL process logic. Moreover, he raised the possibility that investments in existing systems may even be a sunk cost, with no recovery value.

Frazer (1992) suggested that "without the aid of automated tools the effort required to reverse engineer an existing system is likely to be similar to that required for developing a new one" (p. 237). Frazer also said without automated support, the success or failure of a reverse engineering project depends, to a large extent, on human skills.

The interest in automated tools capable of reverse engineering functional design information from legacy systems is based, in part, on the success of reverse engineering tools designed to extract data structures from COBOL source code. Desmond (1992) cited, as an example, a highly successful tool from Bachman Information Systems (Burlington, MA) which provides such capabilities.

Issues in Reverse Engineering

Yourdon (1989b) identified several management issues related to what he identified as RE^3 (reengineering, restructuring, and reverse engineering). A major issue is potential savings, especially with respect to replacing a system rather than trying to prolong its life. Other issues include: (a) how to begin, (b) identifying obstacles, (c) overcoming resistance, the possibility of failure, and (d) the amount of manual effort required.

Yourdon (1989b) also identified technical issues: (a) RE^3 technologies work when expectations are modest, (b) there is no way to turn bad code into good code, bad designs into good designs or bad systems into good systems, and (c) there are major technical difficulties associated with reverse engineering because the mapping from analysis to design, and from design to code, is not a one-to-one mapping.

Importance of the Topic

Worldwide, it has been estimated there are between 90 billion (Connal & Burns, 1993) and 100 billion (Davis, 1991a) lines of legacy system source code in existence. Eighty percent of this source code is written in COBOL (Al-Jarrah & Torsun, 1979; Davis, 1991a; Weinman, 1991). Connal and Burns (1993) estimated the existing systems were written by 2.5 million different programmers and make up 60 million different programs. It has been estimated that the total US investment in existing software is more than $2.3 trillion and the cost of maintaining it is more than $30 billion a year (Davis, 1990). Boehm (1987) estimated the worldwide cost of maintaining software would be $800 billion by the year 2000.

Tilley, Müller, Whitney and Wong (1993) observed that "the software profession has reached a turning point, one where more people are employed to maintain existing applications than to develop new systems from scratch" (p. 142). Britcher and Craig (1986) have identified the problem of upgrading large, complex systems written in

unstructured languages and according to designs that make modification difficult as the major challenge currently facing software system managers.

Rabin (1992) says that achieving competitive advantage in today's international markets demands the efficient use of resources. Two of these key resources are current software systems and the employees who developed them. The systems represent sizable investments in capital. The information system employees have acquired a wealth of information about these production applications and the business principles they support. Therefore, neither the systems nor the knowledge of the employees who developed them can be eliminated without significant resource loss.

Brown (1993) suggested an increasing number of organizations are finding themselves dependent on software written many years ago. A survey conducted by HCS, Incorporated (reported in Weinman, 1991) indicated 80 percent of programmers and analysts in Fortune 1000 companies are engaged in software maintenance activities. A Sentry Market Research survey conducted in 1993 (as reported in Hanna, 1993) indicated maintenance dominates the system developer's time: maintenance activities comprised 43 percent, while new development activities comprised only 31 percent. Volpe and Welty (cited in Weinman, 1991) estimated the world's total resource consumption devoted to the maintenance of existing software systems is more than $120 billion per year. According to Weinberg (1982), the principal mode of software design has

become design by maintenance. He claimed the vast majority of design decisions being put into effect today are created by maintenance programmers, not designers. Maintenance or replacement of legacy application software systems is a growing problem for the data processing industry and is one of the fundamental driving factors for the current interest in reverse engineering and the broader area of reengineering. A recent *Datamation* report (Hayley, Plewa & Watts, 1993) indicated the average chief information officer was involved with 4.4 reengineering projects in 1993 compared with 1.6 such projects in 1992, an increase of 175 percent. Frazer (1992) said "reverse engineering is emerging as one of the most significant developments in the short history of software engineering and the opportunities are immense for those able to provide genuine solutions for very real problems" (pp. 223-224).

There is a close connection between software maintenance and reverse engineering in at least two major areas: (a) Prolonged maintenance of software systems eventually leads to interest in replacing these systems, and (b) it is necessary to understand software to effect both maintenance and reverse engineering. Davis (1991b) estimated nearly half of the typical software maintainer's time is spent analyzing code in an attempt to understand it. It is therefore realistic to expect that reverse engineering techniques can also contribute to extending the life of existing systems by postponing replacement. If maintenance programmers are able to better understand software then the overall cost of maintenance can be reduced, making continued maintenance a more economically viable option.

Significance of the Research

Replacing legacy systems tends to be cost and time prohibitive, but the main obstacle may often be risk. As Ulrich (1990a) noted, time and cost to replace a system, while not insignificant, are secondary to the risk of lost functionality in the replacement system.

Chikofsky and Cross (1990) noted there is a cost associated with understanding software. This cost includes both the time involved in comprehending the software and the time that may be lost because of misunderstanding. The potential cost savings for improving software understanding therefore lies in two areas. The first is in reverse engineering where the primary interest is in reducing the time required to extract some functional level of understanding for the purpose of replacing the old system with a new system. The second is in ongoing maintenance where the primary interest is in reducing both the time required to understand the software and the time lost to misunderstanding. The significant difference between software understanding for system replacement and software understanding for system maintenance is the level of detail and, consequently, the degree of accuracy required. Partee (1993) suggested a clear picture of legacy systems dramatically improves productivity and accuracy by making systems easier to maintain.

Identifying legacy system functions is one of the early considerations in designing replacement systems; it is normally identified as the "current system analysis" or a similar activity. According to Yourdon (1989a), as much as 80 to 90 percent of the functions of a replacement system will be the same as the functions of an existing system. This

probability of functional overlap suggests that it is necessary to review the existing system before developing a replacement system. A current system analysis is needed to ensure that functionality is either included in the replacement system or that the functions that will not be included in the replacement system are eliminated intentionally.

Although design information recovery from legacy systems is an important aspect of systems maintenance as well as systems replacement projects, it has proven extremely difficult to achieve. According to Arango, Baxter, Freeman and Pidgeon (1986), it is not possible to completely recapture the design, but the "approximation error" (the difference between the original and the recaptured design) should be as small as possible.

Definition of Terms

Several key terms must be defined to set the stage for this research. In its relatively short history, reverse engineering has been defined in different ways dependent on the focus or interest of the particular researcher. With the advent of reverse engineering, for example, it was necessary to differentiate between forward engineering and reverse engineering (forward engineering is sometimes called traditional engineering). According to Chikofsky and Cross (1990), *forward engineering* is the process of converting or transforming high-level abstractions and logical implementation-independent designs to the physical implementation of a system. They define *reverse engineering* as the process of analyzing a system to identify its components and their interrelationships and to represent the system in another form or at a higher level of abstraction. Reverse engineering is the

process of transforming or moving from one level of description of a system to another level which is regarded as more abstract or "earlier" in terms of the standard life cycle (Lano, Breuer, & Houghton, 1993). In simpler terms, reverse engineering is the recovery of the original system design, or some parts of the original design, from program source code. In yet simpler terms, reverse engineering can be considered computer "unprogramming."

Brief History of Reverse Engineering

One of the first uses of the term *reverse engineering* occurred in a paper written by M. G. Rekoff (1985) which appeared in the *IEEE Transactions on Systems, Man, and Cybernetics*. Rekoff defined reverse engineering as "developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system" (p. 244). Rekoff noted these specifications are prepared by people other than the original designers without the benefit of original drawings or other documentation, except possibly operations and maintenance manuals. In his view, reverse engineering is just a special case of system engineering. Rekoff believed the goal of reverse engineering is to create a *clone* or to create a *surrogate*. A clone is an exact duplicate of the original article, while a surrogate performs the same function but is not necessarily an exact copy.

Cross, Chikofsky and May (1992) view reverse engineering as a component of a much more comprehensive methodology which focuses on software reuse. In order of application, this structure includes: (a) reverse engineering (includes redocumentation and

design recovery), (b) restructuring, (c) reengineering (includes redevelopment and renovation), and (d) reclamation.

Chikofsky and Cross (1990) suggest the relationship between these elements and the forward engineering methodology are as shown in Figure 1. Note that reverse engineering can take place at a level higher than source code.

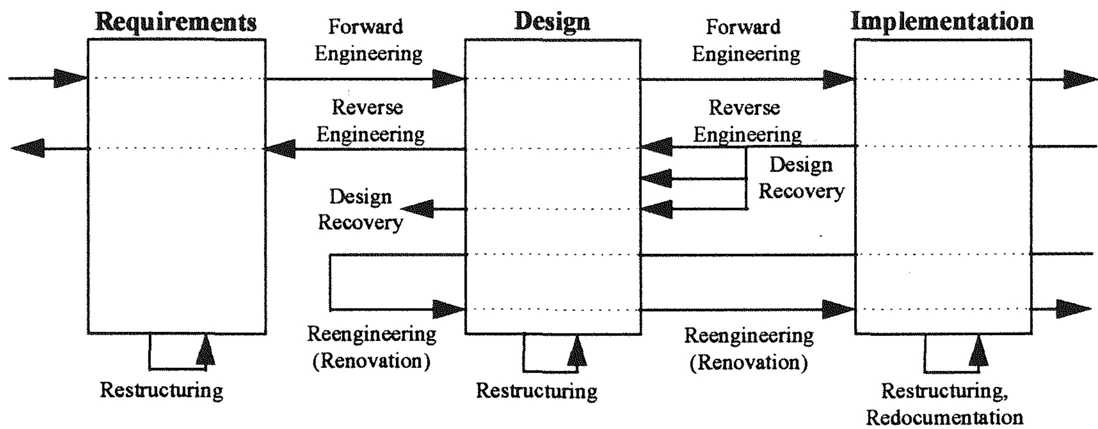


Figure 1. Reverse engineering and related processes are transformations between or within abstraction levels.

Note. Adapted from "Reverse Engineering and Design Recovery, A Taxonomy," by E. J. Chikofsky and J. H. Cross, 1990, *IEEE Software, 7,* p. 14.

Objectives of the Research

The objective of this research was to develop a practical, applied methodology supported by definitive reverse engineering techniques to support the recovery of high-level design information from legacy system COBOL source code. Although there has been extensive

research in reverse engineering, much of the work has been experimental and performed in an academic rather than a real-life environment. Previous research tends to focus on theoretical aspects of reverse engineering rather than on applied concepts. Specific objectives of this research were:

1. To develop a useful, applied approach to high-level design information recovery from legacy system COBOL source code.
2. To support the validity of the approach by reference to relevant theory.
3. To demonstrate feasibility of the approach in a case study.
4. To provide support for the utility of the approach in a case study.
5. To assess the value of the approach for practical application.
6. To form a foundation for future research.

Scope of the Research

A comprehensive approach to this study might have been to examine reverse engineering practices in a number of organizations. Organizations in various business areas that use different programming languages could have been studied. The results of such an approach might reveal some significant new base of knowledge to advance the practice of reverse engineering. However, a broad approach of this nature would raise other issues--such as the ability to generalize results across multiple application domains, programming languages, systems' ages and sizes, criticality of systems, and a host of other factors.

Because a realistic environment for the study was considered essential, a single organization type and programming language was chosen. The author has a long association with the Federal Government, specifically the Department of Defense, and is aware of the problems encountered in developing, maintaining, and replacing computer-based applications in the military environment. The U. S. Air Force's Air Materiel Command was selected as the organization because of the proximity to the author of its headquarters at Wright-Patterson Air Force Base in Dayton, Ohio. This organization was also selected because it relies upon several hundred business-type, computer-based systems to carry out its world-wide logistics support mission. These systems are usually written in the COBOL language (a standard applications programming language in the Department of Defense). Many systems are 20 years old and are expected to be viable for 15 or 20 years into the future (Bennett, 1991). Bennett refers to them as "geriatric" rather than "legacy" systems. These systems, however, are critical to daily operations, represent a significant original development investment, and are difficult to replace because of the excessive costs involved.

Research Questions Investigated

The following research questions were investigated:

1. Can a reverse engineering methodology be derived from an exhaustive study and understanding of the forward engineering process?
2. Can a reverse engineering methodology based on a sound theoretical basis be tested and validated by applying the methodology in a real world environment?

3. Can useful system design information be extracted from legacy system source code using the aforementioned reverse engineering methodology?
4. Will use of a reverse engineering methodology allow analysts to extract essential elements of high-level systems design information from legacy system COBOL source code more efficiently than an unstructured approach?

Research Methodology

This research was composed of five key phases: Approach Selection, Methodology Development, Case Problem Selection, Methodology Application, and Methodology Assessment.

Phase I: Approach Selection

Approach selection involved an examination of the overall domain of reverse engineering; a critical review of the results and limitations of previous and ongoing reverse engineering research; assessing existing reverse engineering techniques; evaluating reverse engineering tools; evaluating the limitations, advantages, practicality, and effectiveness of methodologies relative to the target environment; and determining the fundamental basis of the methodology.

Phase II: Methodology Development

In the second phase of the research the actual reverse engineering methodology was developed. Methodology development was based on identifying techniques and

procedures for implementing the fundamental approach identified in the Phase I. This phase included the identification of relevant source program information such as interfaces with other programs and input and output files. This phase also included developing suitable diagramming techniques and designing a repository for recording recovered design information. The final component of this phase was a methodology for evaluating the viability of a recovered design model to be applied at the end of Phase III.

Phase III: Case Problem Selection

This phase involved the selection of a suitably sized case problem for applying the reverse engineering methodology. System segment or subsystem size, representativeness of programs included, complexity of source code, and similar factors were considered in selecting the case problem.

Phase IV: Methodology Application

Methodology application involved the employment of the reverse engineering methodology developed in the Phase II to an actual problem. For this research, a subsystem of a larger logistics management system typical of the legacy systems found in the U.S. Air Force was used.

The case study approach allowed the methodology to be tested in a "live" environment. Functional and logical documentation was not used during methodology application in order to simulate the unavailability of high-level design information. Extracts of limited

physical design information were used as supporting information during source code analysis. After the methodology application was completed, the resulting system model was informally compared with functional information contained in the high-level documentation.

Phase V: Methodology Assessment

In this phase the results of the methodology application were reviewed to assess the effectiveness of the reverse engineering methodology. Recovered design information was compared with actual design information. Errors or deviations from the actual design were analyzed and assessed to support an evaluation of the methodology effectiveness and to identify possible changes to the methodology

Limitations and Delimitations of the Research

Limitations of the Research

There is one principal limitation of this research: the selection of the system for the test application and reverse engineering methodology assessment. The programs that comprise the selected system are not representative of the complexity of worldwide legacy system programs. Another randomly selected system or a series of randomly selected systems subjected to the same methodology may result in different findings and conclusions.

Delimitations of the Research

The first delimitation of the research concerns the selection of the COBOL programming language. COBOL has certain features that promote better program understanding than other languages. On the other hand, COBOL also includes features that make program understanding more difficult. The results of applying the reverse engineering methodology to other languages, such as FORTRAN, C, or PASCAL, may be significantly different than results achieved with COBOL.

The second delimitation is the operating system environment selected for the case study. The IBM MVS operating environment has certain features (such as a relatively complicated job control language (JCL)) that both contributes and detracts from the effectiveness of recovering design information from source code. A different operating environment (e.g., Honeywell) has a less complex JCL and may have a different effect on the ability to extract design information from source code.

The third and final delimitation is the orientation of the research. While it is well known that both data structure and process structure are equally important in software engineering, this research focused only on the recovery of functional process design information. Although it is not possible to completely avoid the data structure issue in reverse engineering, it was addressed only coincidentally. This delimitation, however, is not felt to have a serious impact on the results of the research because data structure design information is relatively easy to recover from legacy systems.

Contributions of the Research

Theoretical Contribution

Current theories of reverse engineering are primarily based on mathematical concepts. Few reverse engineering theories address its application in a real-life environment. The exploratory nature of this research contributes to qualitative aspects of reverse engineering theory. The primary contribution is an elucidation of how design information is actually extracted from program source code, the sufficiency of recovered data, an assessment of data not present in the source code, identification of missing data, and a scheme for uncovering the missing data.

Managerial Contribution

Primarily, reverse engineering research is presented in academic papers in technical journals. Often, the intent is to present theoretical proofs of a solution to a reverse engineering problem. These papers tend to be difficult to read and understand. Moreover, it is often difficult to see how the research results can be applied to actual problems. In addition, despite the predominant position of COBOL in legacy systems, academic reverse engineering research is often conducted using PASCAL, C, and FORTRAN. This research focused on the COBOL language and addresses real-life problems faced by working technicians.

Academic research on reverse engineering tends to center on automated methods of design recovery by creating assistant-type tools or by applying knowledge-based, or artificial intelligence, techniques. Trade journals, on the other hand, frequently take a simplistic approach to the problem of reverse engineering and lead information system managers to believe that automated tools capable of eliminating the difficulties associated with reverse engineering are, or will soon be, available.

This research fills in the information gap between academic journals and trade magazines with respect to reverse engineering. Focusing on a practical methodology that can be applied by information system practitioners resulted in a teachable, usable, potentially cost saving methodology for information system managers. The availability of this methodology, if it results in only a 0.01 percent reduction in the cost of performing system maintenance, has a potential value of $30 million in the United States alone (0.01 x $30 billion estimated by Davis (1991a) as the cost of maintaining software each year in the US). The value of the reverse engineering methodology to design information recovery oriented toward systems replacement is more difficult to calculate, but probably equally significant.

Retrieval of functional information from legacy system source code could become more effective and efficient. The planning, goal creation, application of techniques, and defined output products for the proposed reverse engineering methodology can replace the “trial and error” approach that is commonly used by inexperienced technicians.

There is a staggering amount of information available related to reengineering and reverse engineering. Reviewing even a small sample of this information is a time-consuming effort. This research contributes to information system professional education and general knowledge by consolidating relevant reverse engineering and peripheral information into a single reference source. This research, in effect, serves as a "bridge" between the theoretical, mathematics-based, academic world and the real-life world of the information systems technician.

Criteria for Success

An essential element for any research project is an objective criteria for determining success. An objective measure must be established to determine when the research has been completed and to evaluate its success. The success of this research is represented by its contribution to the practical interpretation and application of reverse engineering theory in an operational environment. Given the nature of computer software and its complexity, traditional testing and evaluation techniques are difficult to apply to software-related research. The results of traditional evaluation techniques are likewise difficult to interpret. Therefore, the success criteria established for this research, as detailed in the following sections, relied more on practical, demonstrated usefulness than on theoretical evaluation.

Documented Reverse Engineering Methodology

The result of the research is a formal, documented reverse engineering methodology specifically tailored for the types of legacy systems now in operation within the U.S. Air Force Materiel Command. The methodology consists of a series of techniques associated with specific output products that represent the steps to be followed in applying the methodology.

The techniques included in the methodology were based on the results of formal theory and research, or on the results of practical application within the target environment. The specific objective of the methodology was to begin with source code and, as rapidly as possible, reverse engineer a model of the system at a level of abstraction higher than the source code. The methodology supports multiple levels of abstraction. The ultimate goal was to identify a purely functional representation of a legacy system that can be used as the basis for a requirements specification document for a replacement system.

Validated Methodology

The reverse engineering methodology was validated within its intended purpose of recovering functional design information from legacy system source code. The validity of the methodology was demonstrated by a case study which showed the extraction of functional design information from a portion of an actual legacy system. The extracted design information was compared with the functionality of the system as ascertained from documentation.

Program Information Database Conceptual and Logical Data Models

Conceptual and logical data models for a program information data base were developed to support the methodology. The conceptual model is represented as an entity-relationship diagram. The logical model is represented as a table diagram that can be used to implement the database in any relational database management system. A prototype database consisting of major tables and relationships was implemented on a personal computer and used in the test case. The database design was modified as a result of the test case. When coupled with processes to reduce the number of manual steps in the methodology, this database could be the foundation for a reverse engineering assistant tool.

Work Load Estimation Metrics

Four metrics for use in estimating time required to reverse engineer a program were examined. All four were shown to have a linear relationship with reverse engineering analysis time. Two metrics, source lines of code and program complexity, were shown to be the two most reliable. The complexity index was developed as part of the reverse engineering methodology.

Summary

This chapter provided background information on reverse engineering and stated the problem, research objectives, scope, limitations, and contributions. The research

methodology and the success criteria were also highlighted. Chapter II reviews the research literature and the development of reverse engineering practices, procedures, and tools. Chapter II also reviews the literature in three related areas: programming languages, the COBOL language, and program understanding. Chapter III discusses the methodology followed in conducting the research. Chapter IV presents research findings. Chapter V articulates conclusions, implications, recommendations and the summary.

Chapter II

Review of the Literature

Introduction

From a theoretical point of view, there is never a need to reverse engineer a computer-based information system. Software does not wear out nor is it consumed in use; once a system is operational it should run indefinitely. However, there is a significant disparity between this theory and real world application--demonstrated by the urgent need for a reverse engineering methodology to deal with the deteriorating legacy system source code existent in most information systems departments. A review of the literature reveals a gradual shift in interest from programming languages in the 1960s, to software development methodologies in the 1970s, to software maintenance in the 1980s, and to reengineering and reverse engineering in the 1990s. Despite considerable theoretical research in reverse engineering, the practical application of research results remains a problem today.

Organized into nine sections, this chapter provides a basic understanding of reverse engineering, as well as other areas directly related to or impacting reverse engineering.

The first section addresses programming languages. A general understanding of the nature of programming languages, including syntax and semantics, is important for applying reverse engineering.

The second section surveys the COBOL programming language, the language of primary focus in this thesis. The background of the language and some of the major aspects of its syntax are discussed, but the section is not a tutorial on the language.

The third section explores literature related to program understanding. Program understanding is crucial to both software maintenance and reverse engineering. Although information derived by analyzing programs is used is differently in maintenance and reverse engineering, the understanding techniques are the same.

The fourth section scans literature relative to the need for and current interest in reverse engineering.

The fifth section addresses reverse engineering terminology, objectives, components, and problems.

The sixth section reviews representative reverse engineering techniques, methodologies, and tools. Techniques are activities or procedures used in performing reverse engineering, but are not, by themselves, "start-to-finish" approaches. Methodologies are complete

approaches to reverse engineering. Tools are computer-based contrivances designed to implement techniques and methodologies or to support manual reverse engineering efforts. The separation of methodologies and tools, in some cases, is arbitrary; many research tools are based on computer implementations of methodologies.

The final section is a summary of the literature.

Programming Languages

In a simplistic sense, a computer language is a way of instructing a computer to perform useful work for humans. Some early definitions, in fact, reflected this view. Abelson and Sussman (1985), however, contended a programming language is more than just a means of instructing a computer to perform tasks, suggesting a language also provides the framework within which people organize ideas about processes.

Sammet (1972) defined a programming language as a set of characters and rules for combining them which have the following characteristics:

1. Machine code knowledge is unnecessary;
2. There is good potential for conversion to other computers;
3. There is an instruction explosion (from one to many); and
4. There is a notation which is closer to the original problem than assembly language would be.

Pratt (1984) suggested a programming language is any notation for the description of algorithms and data structures. Gopal and Schach (1989) argued "a program can be visualized as an abstract function that generates the output value of the variables based on the specified input values" (p. 133).

Raphael (1966) contended some form of the following components is present in every programming language: (a) an elementary program statement, (b) a mechanism for linking one program statement to another, and (c) a means by which the program can obtain data inputs.

Pratt (1984) stated that programming languages consist of two parts: syntax and semantics. Program syntax is the form in which programs are written. Semantics is the meaning given to the various syntactic constructs.

Cohen (1991) noted meaning is not referred to in formal languages because the main interest is in syntax alone, not semantics or diction. The word "formal" is intended to mean strictly formed by the rules.

Programming Language Syntax

Generally speaking, computer language grammar rules are all syntactic rather than semantic (Cohen, 1991). Pratt (1984) explained computer language grammar as a formal definition of the syntax of a programming language. Grammar consists of a set of

definitions (*rules* or *productions*) that specify the sequences of characters (lexical items) that form allowable programs in the defined language.

Pratt (1984) identified the most prominent syntactic elements of a computer language:

1. The character set - ASCII is common; may limit the input/output devices used.
2. Identifiers - FORTRAN uses 6 characters, COBOL 30.
3. Operator symbols.
4. Key words and reserved words - A *keyword* is an identifier used as a fixed part of the syntax of a statement. IF, THEN, ELSE in COBOL. A keyword is a *reserved word* if it may not also be used as a programmer chosen identifier.
5. Noise words - Optional words which may be inserted in statements to improve readability. COBOL provides many such options. In the statement GO TO <label>, the keyword GO is required, but TO is an optional noise word which carries no information and is used only to improve readability.
6. Blanks - Spaces.
7. Delimiters and brackets - A *delimiter* is a syntactic element used simply to mark the beginning or end of some syntactic unit such as a statement or expression. *Brackets* are paired delimiters, e.g., parenthesis or BEGIN ... END pairs.

Programming Language Semantics

As cited earlier, Cohen (1991) viewed the rules of computer language as being syntactic rather than semantic. Because the syntax of computer languages is much simpler than natural language, there is correspondingly less semantic meaning.

Programming Language Components

Pratt (1984) identified the major components of any computer language:

1. Free and fixed field formats. COBOL uses a combination; procedure division fields tend to be free while working storage fields tend to be fixed.
2. Expressions are the basic syntactic building block. In COBOL, expressions are less important than statements.
3. Statements are the most prominent syntactic component in most languages. Each COBOL statement has a unique structure involving special key words, noise words, alternative constructions, optional elements, etc.
4. Overall program-subprogram structure. Possibilities include separate subprogram definitions, nested subprogram definitions, data descriptions separated from executable statements (as found in COBOL), and unseparated subprogram definitions.

Raphael (1966, pp. 69-70) offered the following hierarchy of program components:

1. Elementary program statements (commands, requirements, or implicit specifications):

   a. Command - An imperative statement that commands the action to be taken without saying anything about what effect will thereby be achieved. The elementary statements of most conventional programming languages are exclusively commands.

   b. Requirement - Describes the effect to be achieved without saying anything about the actions to be taken in achieving the effect nor requiring that programmers know how the effect will be achieved.

c. Implicit specification - Similar to a requirement, but programmers must know something about what actions will be take to achieve the desired effect.

2. Subprogram linkage - Provides a convenient building block to assist programmers in organizing a complex program.

a. Explicit call - The subroutine itself must know how and where to find its arguments, where to put its results, and how to get back to the calling program (e.g., CALL SUBR(Arg)).

b. Execute call - A subroutine call syntactically indistinguishable from the basic instructions of a programming language (e.g., a macro). Rarely used for structure; more often used to eliminate duplicate code sections.

c. Function composition - The mathematical idea of a function carried over into programming to designate a subroutine that calculates a single number, the value of the function.

Job Control Language (JCL)

Dependent on the operating system of a computer, job control language (JCL) is used to control the way computer programs execute, and how they allocate and manage file structures associated with a job. According to Burson, Kotik, and Markosian (1990), the initial problem faced by anyone maintaining or modifying applications controlled by the IBM JCL is to understand data flow among the programs and datasets. JCL Job statements organize groups of programs, Execute statements start specific programs, and DD statements link internal program file names to external hardware devices.

The COBOL Language

The COmmon Business Oriented Language (COBOL) has a history unique among the programming languages developed during the short life span of the computing industry. COBOL affected the way systems were designed and continues to have an impact on the maintenance of the numerous COBOL legacy systems found in government and commercial organizations. A thorough understanding of the nature of COBOL is essential for the reverse engineer working with legacy systems, however, this section only provides highlights of the language.

Lientz and Swanson (1980), in a comprehensive survey of data processing organizations, found COBOL, at 52 percent, to be the most widely used programming language in the United States. A later survey performed by Sentry Market Research (as reported in Keyes, 1992) indicated COBOL is also the most widely used language for maintenance and reengineering (at 65 percent and 51 percent respectively) while C and C++ are noticeably less popular (under 10 percent).

According to Fiorello and Cugini (1984), COBOL is by far the most commonly used language within the Federal government: of approximately 500,000 application software programs, 50 to 60 percent (250,000 to 300,000) are written in some form of COBOL.

COBOL History

In the spring of 1959, a group known as the Conference on Data Systems Languages (CODASYL) held a meeting at the University of Pennsylvania (Cunningham, 1962). The group concluded it would not be possible or practical to apply present or future hardware improvements unless software considerations were given major attention--primarily, the development of a common programming language for all computers. It was proposed that this common language have two or more requirements phases, including a language that was problem-oriented, but machine independent, followed by a general purpose programming language; a language that was systems-oriented and computer independent; systems specifications could be written in a language significant to people as well as machines (Cunningham, 1962).

At that time, the Department of Defense (DOD) was the largest single user of computers. At the urging of the CODASYL group, the DOD agreed to sponsor the common programming language project (Friedman & Cornford, 1989). In May 1959, a meeting was held in the Pentagon (Sammet, 1981); attendees included people from government, the user community, and six computer manufacturers (Gelernter & Jagannathan, 1990). Three main topics were discussed: the time and cost of reprogramming when changing from one computer to another, the inflexibility of programs on simple machines, and the desirability of program interchange (compatibility) between machines (Cunningham, 1962). Data description and statement language task groups were also established.

A COBOL language, COBOL-60, was designed and then implemented by several manufacturers in 1960 (Gelernter & Jagannathan, 1990).

In 1960 the DOD decreed all data processing computers purchased must be supplied with a COBOL compiler (Friedman & Cornford, 1989). Sammet (1981) suggested this edict was responsible for COBOL becoming a de facto standard even before it became an official one.

According to Sammet (1981), who served on one of the language design committees, COBOL was intended from its inception to be used on large (by 1959 standards) computers built for "business data processing," although there was never any real definition of that phrase. Sammett identified the criteria used in the design of COBOL, in decreasing order of importance, as: naturalness, ease of transcripton to required media, effectiveness of problem structure, and ease of implementation.

A revised version of COBOL, COBOL-61, was introduced in 1961 and formed the core of all future versions of the language (Gelernter & Jagannathan, 1990).

COBOL Structure

The COBOL standard is organized in a modular fashion that allows the language to be implemented on a wide range of hardware (Pratt, 1984). The definition is composed of a *nucleus* and a set of 11 *modules*, each of which has between one and three *levels* (see

Table 1). The minimum COBOL implementation consists of the features that make up the lowest level of the nucleus and the modules. Table handling and sequential input/output are the only modules with non-null minimum levels.

Table 1
ANSI-74 COBOL Consists of a Nucleus and 11 Modules

Component	Level 2	Level 1	Minimum level
Nucleus	2	1	
Table handling	2	1	
Sequential I-O	2	1	
Relative I-O	2	1	Null
Indexed I-O	2	1	Null
Sort-merge	2	1	Null
Report writer	2	1	Null
Segmentation	2	1	Null
Library	2	1	Null
Debug	2	1	Null
Inter-program communications	2	1	Null
Communications	2	1	Null

Note. Adapted from "Discussion and Correspondence: A Study of COBOL Portability," by J. M. Triance, 1978, *The Computer Journal, 23,* p. 278.

Because COBOL was originally set up for punched card input (Stern & Stern, 1979), the language was designed using clearly defined fields. Table 2 explains the various fields on a COBOL coding sheet.

Table 2
COBOL Program Field Assignments

Card column	Purpose	Description
1 - 6	Sequence number area	Page (2 positions) and sequential number (4 positions).
7	Continuation field	A hyphen (-) for nonnumeric literals (* is recognized as a comment line).
8-11	A margin (field)	Division, section, and paragraph names begin in the A margin, and must appear on a line with no other entries.
12-72	B margin (field)	All other clauses and statements appear in this area.
73-80	Identification	Area for punched cards; typically used for the program name.

Figure 2 is a sample COBOL coding sheet that shows the field layouts.

COBOL's program organization is monolithic and made up of four divisions (Stevenson, 1975): (a) identification, (b) environment, (c) data, and (d) procedure. These divisions are comprised of structural elements (see Figure 3). According to Pratt (1984) the intent was to separate machine dependent and machine-independent program elements and to separate data descriptions from algorithm descriptions to allow changes in one without affecting the other.

COBOL Coding Form

SYSTEM		PUNCHING INSTRUCTIONS	PAGE OF
PROGRAM		GRAPHIC	CARD FORM #
PROGRAMMER	DATE	PUNCH	

SEQUENCE (PAGE) 1–3	SEQUENCE (SERIAL) 4–6	CONT. 7	A 8	B 12	COBOL STATEMENT 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72	IDENTIFICATION 76 80
	01					
	02					
	03					
	04					
	05					
	06					
	07					
	08					
	09					
	10					
	11					
	12					
	13					
	14					
	15					
	16					
	17					
	18					
	19					
	20					

Figure 2. Sample COBOL coding sheet.

Note. Adapted from *Structured COBOL: American national standard* (p. 9), by V. T. Dock, 1979, St. Paul, MN: West.

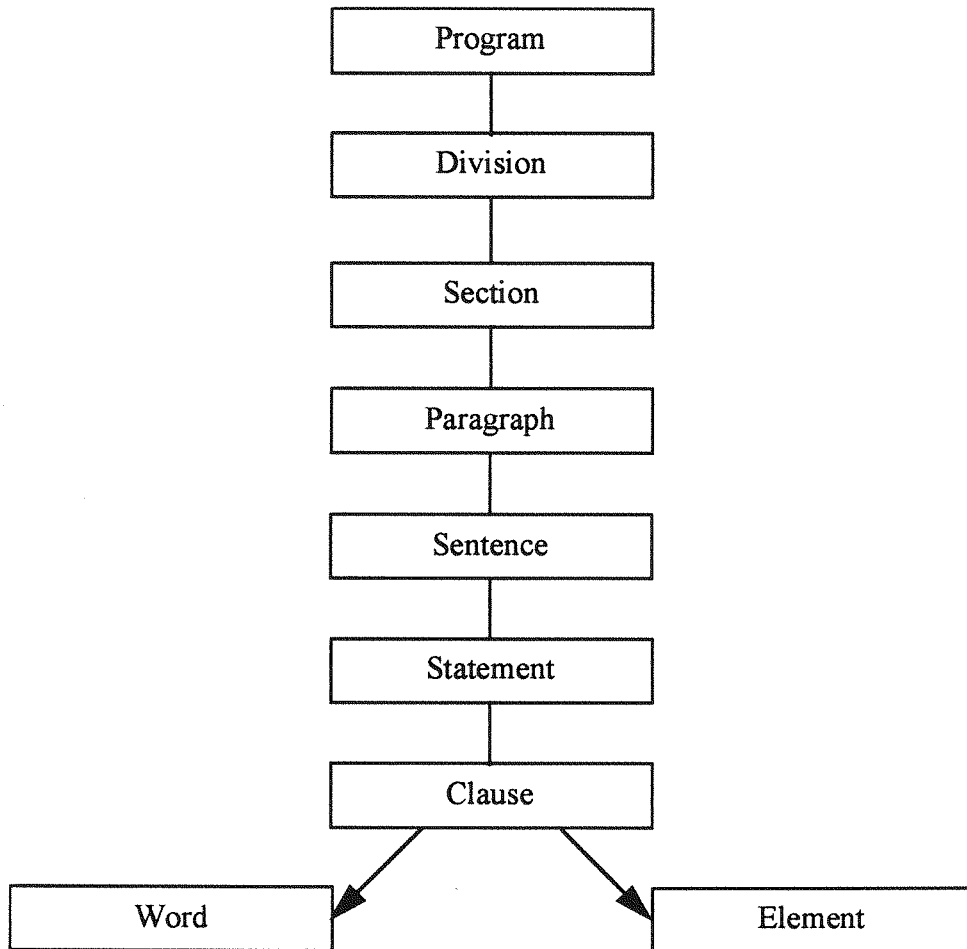


Figure 3. Structural elements of COBOL programs.

Note. Adapted from *The Revolutionary Guide to COBOL* (p. 13), by Y. Handel and B. Degtyar, 1994, Birmingham, England: WROX Press.

Table 3 defines the structural elements of COBOL programs.

Table 3
General Structural Elements of COBOL Programs

Element	Description
Word	Made up of one or more characters.
Clause	Made up of characters or words and specifies an attribute of an entry.
Entry	A group of clauses ending with a period.
Statement	A syntactically valid combination of words and characters that begins with a verb that makes the computer do something (used in the Procedure Division).
Sentence	A sequence of statements, the last of which is terminated by a period and followed by a space. The period is especially important in conditional if/then/else constructs to ensure they are properly terminated.
Paragraph	Consists of one or more sentences. May be executed as a procedure. Paragraph name can be used as a label for GO TO.
Section	Consists of one or more paragraphs. May be executed as a procedure.
Division	Consists of one or more paragraphs or sections.

Note. Adapted from *The Essentials of COBOL I* (p. 27), by R. Cezzar, 1989, Piscataway, NJ: Research & Education Association.

The Identification Division identifies the program to the computer (Handel & Degtyar, 1994). This division contains six paragraphs, of which only the first is required:

```
PROGRAM-ID
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
```

The Identification Division is the least significant of the four divisions because it has no affect on execution; its purpose is to identify a job (program). Stern and Stern (1990) suggested some of the optional entries (such as AUTHOR, INSTALLATION, and

DATE-WRITTEN) provide extremely useful documentation for non-data processing personnel. This reflects the expectation of the original CODASYL group that managers and functional users would read COBOL programs because of its English-like format; it is doubtful this has occurred to any great extent.

The Environment Division describes the computer equipment used by a specific program (Stern & Stern, 1979). It contains paragraphs needed to connect the program with its environment; in particular, it interfaces the data file and device names in the program with the operating environment (Handel & Degtyar, 1994). See Figure 4 for structure delineation.

The Configuration Section contains five paragraphs (of which only the first two are required):

```
SOURCE-COMPUTER.
OBJECT-COMPUTER.
PROGRAM-COLLATING-SEQUENCE.
SEGMENT-LIMIT.
SPECIAL-NAMES.
```

The Input-Output Section links logical program files with physical files on an external device, and contains two paragraphs:

FILE-CONTROL.	Describes the files used in the program.
I-O CONTROL.	This paragraph is only relevant to files that occupy multiple volumes, or single volumes that contain multiple files (such as magnetic tape) (Handel & Degtyar, 1994).

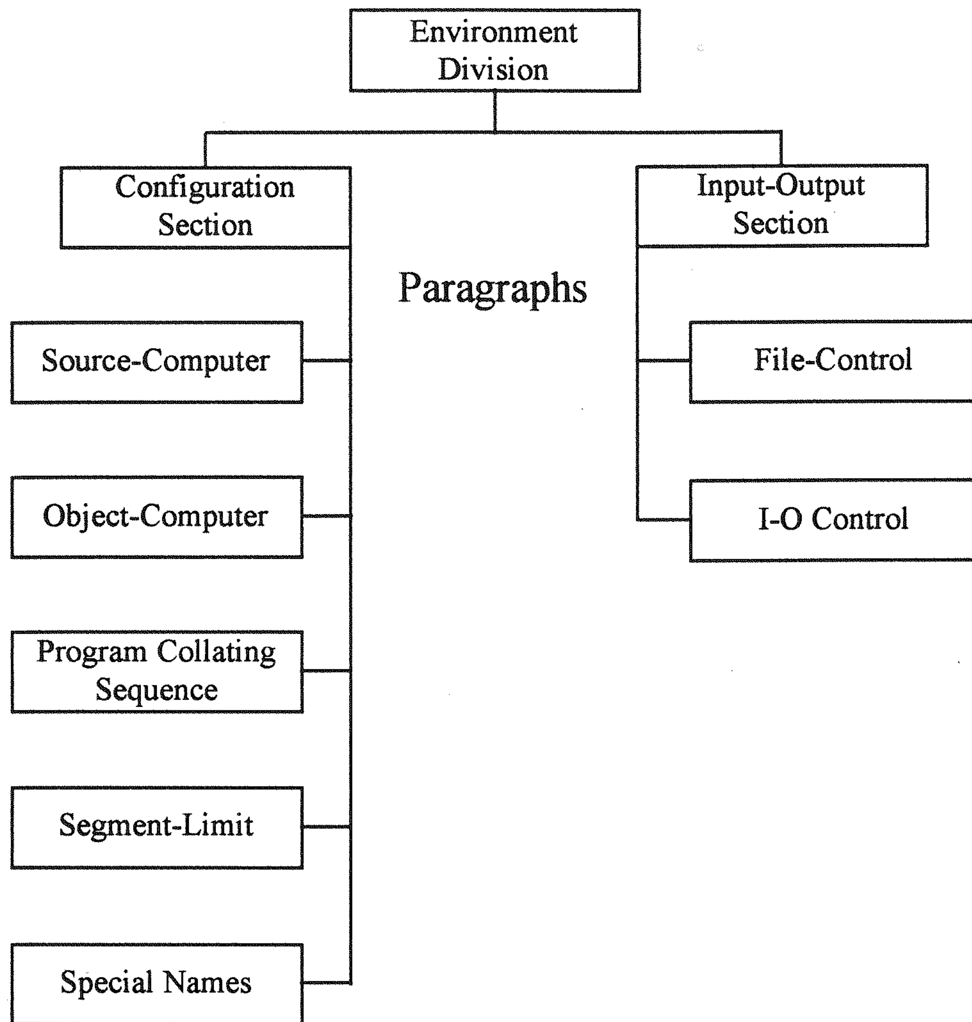


Figure 4. COBOL Environment Division structure.

Note. Adapted from *The Revolutionary Guide to COBOL* (p. 19), by Y. Handel and B. Degtyar, 1994, Birmingham, England: WROX Press.

The Data Division describes input and output formats to be processed by the program, as well as constants and work areas needed to process the data (Stern & Stern, 1979). The Data Division is divided into two sections: (a) the File Section, and (b) the Working Storage Section.

The File Section links program logical files with physical files on external devices and defines all data areas of the input and output files. The Working Storage Section is used to set up memory for fields that are not part of the input or output files.

COBOL data names may be up to 30 characters long. Data names must begin with an alphabetic character and consist of letters, digits, and hyphens. Data names may not begin or end with hyphens, contain embedded blanks, or contain COBOL reserved words.

The general COBOL data organization is made up of files, records, fields, group items, and elementary items (Stern & Stern, 1979):

1. File - The overall classification of data pertaining to a specific category; the major grouping of data containing information of a specific nature; or a major classification of data in a data processing environment.
2. Record - A unit of grouped data within a file that contains information of a specific nature.
3. Field - A group of consecutive storage positions reserved for a specific kind of data.
4. Group Item - A data field that is further subdivided; a major field consisting of minor fields.
5. Elementary Item - A data field not subdivided.

The Procedure Division contains the instructions necessary to read input, process it, and create output. The Procedure Division contains all instructions to be executed; logic is contained within the instructions.

Within the Procedure Division, the traditional paragraph form is terminated by the appearance of the next paragraph name or the End statement (Stevenson, 1975):

```
Paragraph-name.                          Paragraph-name.
   [body of paragraph]         or           [body of paragraph]
Next-paragraph-name.                     END-paragraph.
```

COBOL syntax is designed so programs will be "English-like" or "self-documenting" (Gelernter & Jagannathan, 1990), and is modeled on simple English sentences rather than mathematical expressions. English, as opposed to mathematical notation, is less concise and more varied. The operation *assign variable C the value of variable A divided by variable B* can appear in COBOL as:

```
DIVIDE A BY B GIVING C.
```
or
```
DIVIDE B INTO A GIVING C.
```
or
```
COMPUTE C = A/B.
```
or
```
DIVIDE B INTO A GIVING C REMAINDER D.
```

Nature of COBOL

Control organizing constructs in COBOL are simple: conditional statements, GO TOs, and a looping construct called the PERFORM statement. The PERFORM statement supports both bounded and unbounded (FOR loop and WHILE loop style) iterations (Gelernter & Jagannathan, 1990).

COBOL is described as basically a constructive language, but it includes highly declarative elements: (a) the range of values a variable can take is specified by the PICTURE declaration; and (b) active typing (e.g., the MOVE CORRESPONDING statement) can cause a substantial amount of activity to occur implicitly (Gelernter & Jagannathan, 1990).

Al-Jarrah and Torsun (1979) performed a static analysis of 340 COBOL programs collected from commercial and industrial installations. Program sizes ranged from 50 to 5,000 lines of code. The average COBOL program was 666 source cards. The average length of user defined names was 7.81 characters (out of a possible 30 characters). The average number of files was 3. The USAGE clause was not specified in 80.5 percent of all data items (defaulting to DISPLAY and reflecting the preponderance of non-computational data in commercial computing). The OCCURS clause is used to declare arrays; its low frequency (14.9 percent) reveals arrays are not widely used by COBOL programmers. Five verbs (MOVE, IF, GO TO, PERFORM, and ADD) accounted for 84 percent of the verbs. The total frequency of comments (NOTE xxx and * in column 7) was 3.35 percent.

In a survey of 100 representative on-line and batch programs from the German commercial sector, Sneed and Jandrasics (1987) reported:

1. Programs consisted of one module [a separately compilable and testable unit].
2. All sections were connected via common data in the DATA DIVISION.
3. Programs were well-structured.
4. Programs contained data which were not used at all.

5. The average number of data definitions was 1104.
6. The average size of the PROCEDURE DIVISION was 2255 lines.

Figure 5 is a sample of a typical batch-oriented COBOL program.

COBOL Disadvantages

Hicks (1975) noted a problem with the COBOL looping construct because it only allows an exit from the top. Hicks maintains the typical loop in business programming does not have an exit at the top or the bottom, but somewhere in between.:

Goguen (1975) noted the PERFORM . . . THRU format of the PERFORM statement is a hazard in providing a means to "fall through" from one paragraph to another.

One of the objectives in developing the COBOL language was to make it possible for nonprofessionals to write programs. However, Weinberg (1971) suggested this was not necessarily a good objective because COBOL allowed nonprofessionals to write programs. The quality of programs produced in the mid 1960s and early 1970s by these inexperienced programmers contributed to the problem of dealing with legacy systems today.

```
000001      IDENTIFICATION DIVISION.
000002      PROGRAM-ID.  DELINRPT
000003      AUTHOR.  JOHN SMITH.
000004      INSTALLATION.  ACME PAINT, INC.
000005      REMARKS.  THIS PROGRAM PREPARES A REPORT BY NAME OF THOSE
000006      PATRONS WHOSE CONTRIBUTIONS WERE BELOW TARGET.
000007      INPUT CONTAINS THE PATRON NAME, TARGET CONTRIBUTION,
000008      ACTUAL CONTRIBUTION AND DATE OF CONTRIBUTION.
000009      DATE-WRITTEN.  JANUARY, 1970.
000010      DATE-COMPILED.  JANUARY, 1970.
000011      SECURITY.  UNCLASSIFIED
000012      ENVIRONMENT DIVISION
000013      CONFIGURATION SECTION.
000014                  SOURCE-COMPUTER.  IBM-370.
000015                  OBJECT-COMPUTER.  IBM-370.
000016                  PROGRAM-COLLATING-SEQUENCE.
000017                  SEGMENT-LIMIT.
000018                  SPECIAL-NAMES.
000019      INPUT-OUTPUT SECTION.
000020      FILE-CONTROL.
000021                  SELECT PATRON-FILE
000022                        ASSIGN TO SYS006-UT-2400-S.
000023                  SELECT DEFICIENCY-LIST
000024                        ASSIGN TO SYS009-UR-1403-S
000025      DATA DIVISION.
000026      FILE SECTION.
000027      FD PATRON-FILE
000028                  RECORD CONTAINS 74 CHARACTERS
000029                  LABEL RECORDS ARE STANDARD.
000030
000031      01 PATRON-RECORD.
000032            05 PR-NAME.                  PIC X(18).
000033            05 FILLER                    PIC X(42).
000034            05 PR-TRGT-CON               PIC 9(4).
000035            05 PR-ACTL-CON               PIC 9(4).
000036            05 PR-CON-DATE
000037                  10 PR-CON-MONTH        PIC X(2).
000038                  10 PR-CON-DAY          PIC X(2).
000039                  10 PR-CON-YEAR         PIC X(2).
000040
000041      FD DEFICIENCY-LIST.
000042            RECORD CONTAINS 132 CHARACTERS
000043            LABEL RECORDS ARE OMITTED.
000044
```

Figure 5. Sample COBOL program.

```
000045      01 DEFICIENCY-LINE.
000046              05 FILLER                PIC X.
000047              05 DL-NAME               PIC X(18).
000048              05 FILLER                PIC XX         VALUE SPACES.
000049              05 DL-CON-MONTH          PIC XX.
000050              05 FILLER                PIC X          VALUE "/".
000051              05 DL-CON-DAY            PIC XX.
000052              05 FILLER                PIC X          VALUE "/".
000053              05 DL-CON-YEAR           PIC XX.
000054              05 FILLER                PIC X(4)       VALUE SPACES.
000055              05 DL-TRGT-CON           PIC 9(4).
000056              05 FILLER                PIC XXX        VALUE SPACES.
000057              05 DL-ACTION-CON         PIC 9(4).
000058              05 FILLER                PIC XXX        VALUE SPACES.
000059              05 DL-AMT-DEF            PIC 9(4).
000060              05 FILLER                PIC XXX        VALUE SPACES.
000061              05 DL-DEF-PERCENT        PIC 99.9.
000062              05 FILLER                PIC X          VALUE '%'.
000063              05 FILLER                PIC X(73)      VALUE SPACES.
000064
000065      01 TOTAL-LINE.
000066              05 FILLER                PIC X.
000067              05 TL-DEF-PATRONS        PIC 999.
000068              05 FILLER                PIC X(38)      VALUE SPACES.
000069              05 TL-AMT-DEF            PIC 9(6).
000070              05 FILLER                PIC X(81)      VALUE SPACES.
000071
000072      WORKING STORAGE SECTION.
000073
000074      01 WS-SWITCHES.
000075              05 WS-EOF-SWITCH         PIC XXX.
000076
000077      01 WS-ARITHMETIC-WORK-AREAS.
000078              05 WS-AMT-DEFICIENT      PIC 9(4).
000079              05 WS-TOTAL-AMT-DEF      PIC 9(6).
000080              05 WS-DEF-FRACTION       PIC V999.
000081              05 WS-DEF-PERCENT        PIC 99V9.
000082              05 WS-DEF-PATRON         PIC 999.
000083
000084      PROCEDURE DIVISION.
000085
000086      000-PRINT-DEFICIENCY-LIST.
000087                      OPEN INPUT PATRON-FILE
000088                      OUTPUT DEFICIENCY-LIST.
000089                      PERFORM 100-INITIALIZE-VARIABLE-FIELDS.
000090                      READ PATRON-FILE
000091                              AT END MOVE "YES" TO WS-EOF-SWITCH.
000092                      PERFORM 200-PROCESS-PATRON-RECORD
000093                              UNTIL WS-EOF-SWITCH IS EQUAL TO "YES".
```

Figure 5. (continued)

```
000094                  PERFORM 700-PRINT-TOTAL-LINE.
000095                  CLOSE PATRON-FILE
000096                            DEFICIENCY-LIST.
000097                  STOP RUN.
000098
000099         100-INITIALIZE-VARIABLE-FIELDS.
000100              MOVE "NO" TO WS-EOF-SWITCH.
000101              MOVE ZERO TO WS-TOTAL-AMT-DEF,
000102                  WS-DEF-PATRONS.
000103
000104         200-PROCESS-PATRON-RECORD.
000105              IF PR-ACTL-CON < PR-TRGT-CON
000106              PERFORM 210-PROCESS-DEFICIENT-PATRON.
000107              READ PATRON-FILE
000108                  AT END MOVE "YES" TO WS-EOF-SWITCH.
000109
000110         210-PROCESS-DEFICIENT-PATRON.
000111              MOVE PR-NAME TO DL-NAME.
000112              MOVE PR-TRGT-CON TO DL-TRGT-CON.
000113              MOVE PR-ACTL-CON TO DL-ACTL-CON.
000114              MOVE PR-CON-MONTH TO DL-CON-MONTH.
000115              MOVE PR-CON-DAY TO DL-CON-DAY.
000116              MOVE PR-CON-YEAR TO DL-CON-YEAR.
000117              SUBTRACT PR-ACTL-CON FROM PR-TRGT-CON
000118                  GIVING WS-AMT-DEF.
000119              MOVE WS-AMT-DEF TO DL-AMT-DEF.
000120              DIVIDE PR-ACTL-CON BY PR-TRGT-CON
000121                  GIVING WS-DEF-FRACTION.
000122              MULTIPLY WS-DEF-FRACTION BY 100
000123                  GIVING WS-DEF-PERCENT.
000124              MOVE WS-DEF-PERCENT TO DL-DEF-PERCENT.
000125              WRITE DEFICIENCY-LINE
000126                  AFTER ADVANCING 2 LINES.
000127              ADD WS-AMT-DEF TO WS-TOTAL-AMT-DEF.
000128              ADD 1 TO WS-DEF-PATRONS.
000129
000130         PRINT-TOTAL-LINE.
000131              MOVE WS-DEF-PATRONS TO TL-DEF-PATRONS.
000132              MOVE WS-TOTAL-AMT-DEF TO TL-TOTAL-AMT-DEF.
000133              WRITE TOTAL-LINE
000134                  AFTER ADVANCING 3 LINES.
```

Figure 5. (continued)

Note. Adapted from "Function Recovery Based on Program Slicing," by F. Lanubile and G. Visaggio, 1993, *Proceedings of the IEEE Conference on Software Maintenance,* p. 401.

Pratt (1984), Gelernter and Jagannathan (1990), Price, et al. (1993), and Markosian, Newcomb, Brand, Burson, and Kitzmiller (1994) identified numerous problems with the COBOL language:

1. COBOL has no statement bracketing and no local variables.
2. COBOL has no subroutines.
3. COBOL source programs are bulky and verbose. Programmers must generate more text than required in other languages.
4. COBOL allows implicit programming through type coercion. Conversion routines can be invoked implicitly and automatically simply by writing an assignment statement that involves variables with different formats.
5. COBOL has no procedures.
6. There are no subprograms that accept parameters. There is a standard definition for a procedure construct, and it may be implemented "optionally" in extended versions of the language (the COBOL ANSI standard developed in 1974).
7. Names declared in the data division are global to the program.
8. A COBOL paragraph may be invoked from many places in different ways.
9. The working storage section is like an all-engulfing common area, making it very difficult to limit side effects. COBOL variables are global. Any program component can directly access any variable field.
10. The PERFORM verb (the heart of COBOL control flow) has many formats.

Hennell, McNicol, and Hawkins (1980) and Lano, et al. (1993) offered additional COBOL problems:

1. COBOL uses data formats instead of data types.
2. COBOL allows the mixed use of PERFORM statements and the ordinary fall-through execution of paragraphs.
3. Some versions of COBOL include unstructured constructs (i.e., GO TO and ALTER GO TO).
4. COBOL has too many specialized verbs and variant verbs, the exact semantics of which requires specialized programmer knowledge.
5. In COBOL a line may contain multiple statements, making it difficult to label jump-from and jump-to points.
6. Detection of the choice clause may be made difficult if the THEN keyword is absent.
7. The SEARCH statement, which has a logical structure not found in other high-level languages, may make the identification of a choice clause difficult.

Ricketts, DelMonaco, and Weeks (1989) claimed data flow analysis for COBOL is more difficult than many other languages because:

1. Actions on elementary data elements affect group items.
2. Actions on group items affect elementary items.
3. Multiple conflicting definitions for the same physical storage locations are common (the REDEFINES clause, for example).

4. COBOL is not a strongly typed language; all data definitions are global. Unintended side effects are often inherent in old systems.

Ricketts, et al. (1989) pointed out that in COBOL, code analysis (i.e., the discovery of data definitions, flows, and rules, wherever they exist in source code) includes file and record declarations in Environment and Data Divisions, data flows within programs and between files in the Procedure Division, and data flow between programs in the Linkage Section and in JCL data definition (DD) statements.

COBOL Advantages

Gelernter and Jagannathan (1990) suggested the record structure features of COBOL are an advantage because it is possible to "isolate a piece of the broader naming environment, enclose it in a wrapper, give it a name, and treat it as a single (compound) unit within the top-level environment" (p. 161).

Lano, et al. (1993) noted COBOL data declarations provide information about program data structure and associate records and record fields to files.

Program Understanding

Program understanding is the process of reading a program or a series of programs in a system for the purpose of extracting the semantic content; in most cases, the target program has been written by someone other than the reader. This process is required in

both maintenance and reverse engineering. Standish (1984) estimated 50 to 90 percent of maintenance time is devoted to program comprehension.

Ourston (1989) argued that computer automatic program recognition (understanding) is similar to natural language research. Although computer program languages are more structured that natural language, there is an infinity of possible expressions and possible interpretations.

As Sage (1993) noted, it is difficult to describe any large and complex system in terms of any of the three fundamental dimensions of *structure*, *function* or *purpose* because these dimensions are neither mutually exclusive, nor collectively exhaustive. In the case of a computer system, the understanding process of these dimensions is predicated on understanding the individual programs that comprise the system.

Dietrick and Calliss (1992) used the term *code analysis* and described it as a generic term denoting programmer activities where the primary emphasis is on examining a piece of program code. Two important aspects are: (a) determining dependencies between program components, and (b) analyzing program component use.

Kozaczynski, Letovsky and Ning (1991) identified three understanding-intensive tasks related to software:

1. Validation and Verification - Given a piece of code, verify the functional behavior meets its specification.
2. Maintenance - As the need for software understanding always occurs on an "as needed" basis, the maintainer obtains the minimum information necessary to make a change.
3. Reuse - Answers the question, "What does a given component do?" Can occur within a system, between systems, or between a system and a library of reusable components; reuse can occur at the code component or abstract design level.

Maintenance, reverse engineering, and reuse are based on the ability to recognize, comprehend, and manipulate design decisions in source code (Rugaber, Ornburn, & LeBlanc, 1990). A current software engineering challenge is developing technology to make old software systems more comprehensible (Kozaczynski, et al., 1991). As Berns (1984) proposed, "Program maintainability and program understanding are parallel concepts: the more difficult a program is to understand, the more difficult it is to maintain" (p. 14).

Scherlis (1984) discussed the need to understand the *causal* connections between software requirements and the computer programs that realize them and compared this understanding with the causative nature of mathematical logic to mathematical reasoning.

Biggerstaff, Mitbander, and Webster (1994) discussed the *concept assignment problem.* They defined this as the problem of discovering the human-oriented concepts of computational intent (through a process of analysis, experimentation, guessing, and crossword puzzle-like assembly) and assigning them to their realizations within a specific program or its context.

The Meaning of Program Understanding

Martin and McClure (1983) defined understandability as the ease with which the function of a program and how it achieves this function can be understood by reading the program and its documentation. Martin and McClure claimed an understandable program allows a reader to determine the program objectives, assumptions, constraints, inputs, outputs, components, relationships to other programs, and status.

Choi (1993) said "Understanding a program involves assigning meaning to a program text, more meaning than is literally there" (p. 40). Robson, Bennett, Cornelius, and Munro (1991) echoed Choi's view: "Comprehension involves applying the [program] syntactic knowledge to develop an internal semantic representation" (p. 80).

Chen, Heisler, Tsai, Chen, and Leung (1990) defined program understanding in terms of maintenance: "One cannot maintain a program unless one understands it. Program understanding can be a complex task for large applications. Program understanding often involves the specification, the design and the code as well as the interrelationships between them" (p. 4).

Berns (1984) viewed a program as a set of static definitional statements and a set of executable statements; definitions establish the attributes and interrelationships of certain program elements, such as symbolic names. Berns suggested program understanding involves understanding how the dynamic portion of a program manipulates and controls the static elements.

Biggerstaff, et al. (1994) said program understanding is achieved when it is possible to explain the program, its structure, its behavior, its effects on operational context, and its relationship to its application domain. This explanation takes place in a form much different than that used to construct the program.

Software Psychology

Weiser and Shneiderman (1987) defined software psychology as "the study of human performance in using computer and information systems" (p. 1399). Program understanding falls into the category of using computer and information systems, and is a subject of interest to software psychologists. An abundance of literature on the process of programming and program understanding is found in the study of software psychology.

Shneiderman (1980) identified psychological complexity as a factor related to program comprehension. He defined psychological complexity as "characteristics which make it

difficult for humans to understand software" (p. 67). Shneiderman suggested program complexity can be logical, structural, or psychological:

1. *Logical complexity* involves program characteristics that make a proof of correctness difficult, long, or impossible (due to the number of distinct possible program paths).
2. *Psychological complexity* (comprehensibility) refers to characteristics which make it difficult for humans to understand software (e.g., the number of IF statements, module size, and the number of non-normal exits from a decision statement). This factor can also be influenced by structural and logical complexity or other factors such as code comments and external documentation.
3. *Structural complexity*. There are two elements in this factor:

   a. *Absolute structural complexity* is a measure of the number of modules that make up a program (Stevens, Constantine & Myers, 1974; cited in Shneiderman (1980)). A *module* is defined by Dietrick and Calliss (1992) as a named collection of program components where a programmer has control over the program components that are imported from or exported to the surrounding environment. A program is made up of a hierarchy of modules, consisting of instructions, data, and the underlying execution control mechanism (Tian & Zelkowitz, 1992).

   b. *Relative structural complexity* is the ratio of the number of module linkages to the number of modules.

Weiser and Shneiderman (1987) discussed the semantic knowledge of application domain and programming concepts necessary for software understanding.

1. *Semantic knowledge of application domain*. The programmer's knowledge of some field or application area. This knowledge is independent of the computer implementation and is level structured (low-level, mid-level, and high-level).
2. *Semantic knowledge of programming concepts*. The programmer's knowledge of programming practices, algorithms, file structures, data structures, programming language features, operating systems, etc. This knowledge is independent of a particular program's application domain.
3. *Syntactic knowledge*. The details about how to express a semantic knowledge concept in a programming language. Syntactic knowledge is language dependent, arbitrary, requires rote memorization, and is forgotten unless frequently rehearsed.

Factors Affecting Program Understanding

Sage (1977) noted a common difficulty in comprehending a complex system is no one person has enough knowledge to develop a complete set of descriptive elements. Data about a complex system is often incomplete or faulty. The model structure of the complex system may also be unverified or incomplete.

Program structure is the organization and expression of program logic (Boehm-Davis, Holt, & Schultz, 1992). Miller and Strauss (1987) claimed a structured program is better than an unstructured one because it is easier for a programmer to bound (e.g., in analyzing a routine, the programmer need not be concerned with the invoker or any routine invoked by the module under analysis).

Van Zuylen and Estdale (1993) defined program comprehension as the construction of a multi-level, multi-view, representative internal semantic structure. This process is described as follows:

1. A software engineer must understand a program's internal semantic structure.
2. Most semantic knowledge can be derived from source code.
3. Most programs contain a mixture of source code, an interaction with the environment (e.g., a DBMS), a transaction processing system, and calls to external procedures which are external to the program (i.e., user interface libraries).
4. The semantics are found by combining information from language semantics with information from software environment documentation and library specifications. Software engineering knowledge is stored partly in the mind of the software engineer and partly in the software environment documentation.
5. The source code is interpreted; the semantics are determined by transforming information from source code and software engineering knowledge into a representation of the program's semantics.

Program documentation, especially external documentation, is an important contributor to program understanding if it is well written and kept up to date. Younger (1993), explained program (or system) documentation as anything that provides information about a software system including source code, JCL, test data, developed documents, user documentation and code analysis results.

Van Zuylen (1993) said documentation can be considered a collection of different views of a system. Some low-level technical documentation can be extracted from source code (e.g., flow charts and cross reference tables). However, van Zuylen claimed higher level documentation that represents the design and specification of a program cannot be derived completely automatically; human intervention is necessary.

According to Grumman and Welch (1992), documentation rarely corresponds completely to the current state of the software even in new applications where a formalized, structured development method was used.

Weinberg (1971) listed four program aspects that may impact understanding:

1. Machine limitations - A programmer may include coding to overcome machine limitations, but it is rarely explicitly marked. One area where machine limitations are rife is intermediate storage.
2. Language limitations - Some languages are more suitable for a particular application: FORTRAN for scientific and engineering applications, COBOL for business applications. Using an inappropriate language can inhibit program comprehension.
3. Program limitations - Some code may have been written merely because the programmer did not have complete knowledge of the computer or the language.
4. Historical traces - Some pieces of code may have been written for obscure historical reasons.

Gillis and Wright (1990) proposed high-level comprehension of existing source code is becoming more difficult to achieve as systems increase in overall size and complexity; much of the time spent trying to comprehend source code is not productive because either initial text design documents are not clearly representative of what was coded or post-coding documentation is not current.

Naming conventions can also affect a program's understandability. As Miller and Strauss (1987) noted, poor data and procedure names inhibit program understanding. Moreover, inconsistent names for fields used in multiple programs make program understanding more difficult by preventing knowledge about one program from being transferred to another.

Teasley (1994) tested the hypothesis that poor program naming style affects comprehension of function, but not other types of comprehension. Results of the experiment did not support the hypothesis for expert programmers, but it did support the hypothesis for junior programmers.

Elshoff and Marcotty (1982) indicated program readability depends on the reader's familiarity with programs, knowledge of the application area, and individual programming style; these are independent of the program.

Domains and Program Understanding

Hall (1992) described domain analysis as the process of acquiring understanding of an application area. Layzell and Macaulay (1994) described domain knowledge as referring to the knowledge of working practices within the organization, knowledge of the organization's business functions and knowledge of the organization's computer systems.:

Brooks (1983) described the programming process as one of constructing mappings from the problem domain, through intermediate domains, and into the programming domain. Comprehending a program involves reconstructing part or all of these mappings.

Kozaczynski and Wilde (1992) illustrated the importance of domain concepts as shown in Figure 6. Note the domain shift (conceptual leap) that occurs between the logical objects and the first components of the implementation domain. Kozaczynski and Wilde argued this shift is one reason for difficulty in reverse engineering.

Kozaczynski, Ning, and Engberts (1992) described a program as containing language concepts and abstract concepts. Language concepts are syntactic entities (e.g., variable declarations, modules, and statements) defined by the syntax of a programming language. Abstract concepts are language-independent ideas of computation and problem solving methods (e.g., programming concepts, architectural concepts, and domain concepts.

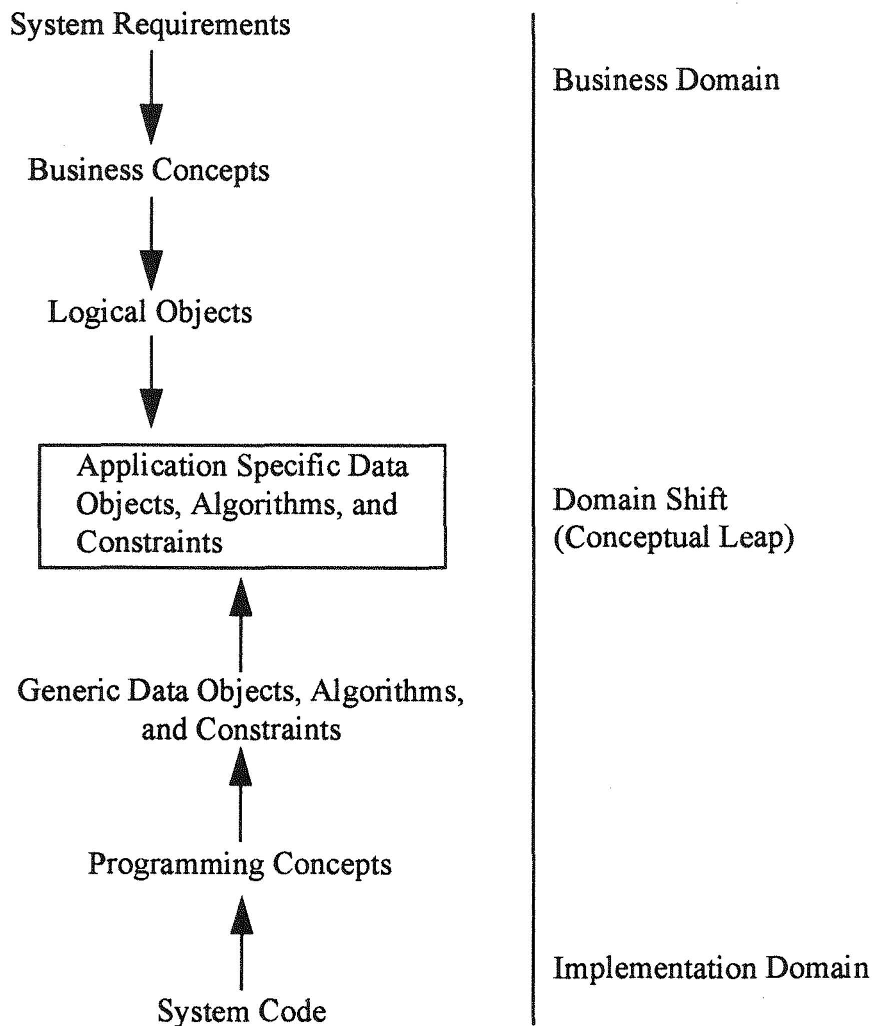


Figure 6. System forward engineering and reverse engineering.

Note. Adapted from "On the Reengineering of Transaction Systems," by W. Kozaczynski and N. Wilde, *Journal of Software Maintenance: Research and Practice, 4*, p. 148.

Programming concepts are general coding strategies, data structures, and algorithms. Architectural concepts are associated with interfaces to execution environment components (e.g., operating systems, transaction monitors, networks, and the databases). Domain concepts are application or business logic functions implemented in code.

Winograd (1979) identified three description domains for complex systems: subject, interaction, and implementation. Each domain is appropriate (and necessary) for understanding some aspect of the system, i.e.:

1. Subject Domain - Descriptions of objects (e.g., buildings, rooms, courses, departments) and processes (e.g., the scheduling of events).
2. Domain of Interaction - Relevant objects take part in the system's interactions (e.g., users, files, forms, maps, statistical summaries). Processes include querying the system, scheduling a new event, and proposing a schedule for a new quarter.
3. Domain of Implementation - The objects in this domain include everything from individual memory lists and subroutines to subsystems (e.g., the file system, the memory management system, the operating system), running processes, hardware devices and code segments.

Approaches to Program Understanding

Bush (1993) looked at mathematical representation as a way of understanding programs. Bush said the most useful mathematical formalism for representing the semantics of

computer programs is graph theory--the study of nodes (computational statements) and the connections between them (control flow statements).

Harandi and Ning (1988) suggested programs can be viewed from four levels of detail in increasing order of abstraction: (a) implementation, (b) structure, (c) function, and (d) domain. Steps to reach each of these levels, working from lowest level to highest level, are:

1. Implementation Level - Remove program language and implementation specific features. Understanding at this level requires knowledge of language syntax and semantics.
2. Structure Level - Further abstract language dependent details to show details of program component dependencies.
3. Function Level - Relate pieces of program to their functions to reveal logical (versus syntactical or structural) relationships.
4. Domain Level - Abstract further by replacing the program's algorithmic nature with concepts specific to an application domain.

Biggerstaff (1989) identified questions a software engineer asks when trying to understand a system:

1. What are the modules? Some languages formalize the notion of a module; program structures are associated with informal semantic concepts to create semantically rich

natural language abstractions (conceptual abstractions) representing the essential concept underlying the module.

2. What are the key data items? What abstract, informal concepts do they relate to? What is their relationship to previously identified modules?
3. What are the software engineering artifacts? These can include problem description language (PDL), dataflows, module refinements and a data dictionaries.
4. What are the informal design abstractions? These are expressed in natural language prose.
5. What are the relationships of design abstractions to code? Data flow diagram segments are associated with implementation code. A set of organized structures are established to help understand code-oriented details.

The Need for Reverse Engineering

Tamai and Torimitsu (1992) surveyed 95 software applications in 1991 and measured the age of the software when it was replaced. The average lifetime was reported to be 10.1 years, with a maximum and minimum of 30 and 2 years respectively; the standard deviation was 6.2 years. Generally, they found that small-scale software tends to have a shorter life and the age of software systems is approximately the same regardless of application area.

Arango, Baxter, Freeman and Pidgeon (1985) noted that the original design for most software is inaccessible because the original requirements analysis and specifications, if

recorded, are out of date. The existing software usually contains implicit assumptions about the environment, but design and environment information recorded in documents cannot be automatically processed.

Pfrenzinger (1992) said existing systems have turned into the Achilles' heel of information system departments. Many aging systems are the backbone of a company's critical production processing, but they are difficult to change, expensive to replace, vulnerable to many problems, and are impossible to understand. Their documentation is outdated and useless. Aging systems consume 75 percent of the information system budget. As organizations move in new directions, legacy systems can't be made to follow.

Legacy Systems

Atkins (1994) suggested legacy systems were originally designed as transaction processing machines to help run operations, not as decision support engines. Therefore, systems are now incapable of satisfying the information requirements of the organizations they support.

Welch and Grumann (1993) reported the cost of adding new functions to old systems increases dramatically, while response times to implement such changes increases disproportionally. They suggested the impact of legacy systems on data processing budgets is significant, requiring 50 to 90 percent of maintenance resources. Moreover, Welch and Grumann said this represents the cost of standing still.

Ning, Engberts, and Kozaczynski (1994) said many large companies are facing the problem of legacy systems inhibiting business growth and capacity to change. Limited options for dealing with legacy systems are available:

1. Develop a new system to replace the legacy system. The legacy system may contain critical business rules that are assets to the organization, but those embedded in the code may not be accurately and explicitly documented.
2. Encapsulate a legacy system to allow it to be used as a whole under a new execution environment or within a new system.
3. Recover reusable components from the legacy system.

Holloway (1992) supported this notion as well. Holloway said all information technology sites have major investments in software applications in terms of code and data structure and these are an irreplaceable corporate asset.

Lenihan (1993) characterized legacy systems as typically more than seven years old, using outmoded or unique technologies, having ineffective reporting systems, and poorly structured program code, and using system and human resources inefficiently.

Mattison (1993) observed that legacy systems are not just a *part* of business, they *are* the business because they define how people do their jobs, how they communicate, and how they relate to each other. In this respect, legacy systems describe the infrastructure of the

corporation; they are tools, the end product of years of work and effort by hundreds of dedicated people. Rather than being useless, outmoded or wasteful, they are essential to an organization's existence.

Hickey and Jennings (1994) described a typical legacy system in an auto insurance company that consists of more than 2 million lines of COBOL, ALC, and PL/I code. Programs average about 1,000 lines of code (with many exceeding 5,000 lines), and are complex, poorly structured, and undocumented. The programs date to 1975 and have been maintained by 100 different programmers. Since original developers and users have moved on, no one person really understands the system. An average of 180 maintenance and enhancement projects are implemented on the system each year.

Software Aging

Boehm (1981) summarized Lehman's (1980) first two laws of large program evolution:

1. Continuous Change - A large program being used undergoes continuous change or it becomes progressively less useful. Boehm's comment was that all large programs have a non-trivial investment segment.
2. Increasing Complexity - As a large program is continuously changed, its complexity increases unless work is done to maintain it.

Beck and Eichmann (1993) suggested long-lived components frequently accumulate substantial functionality over their lifetimes--the kitchen sink syndrome. As more

functions are added, the comprehension required for modification or reabstraction becomes increasingly difficult.

Welch and Grumman (1993) said systems, in general, become more complicated and less manageable the more they are changed, modified, or extended. This is particularly true in data processing systems because of the nature of most computer languages.

Even successful software inevitably evolves and the process of evolution leads to degraded structure and increasing complexity unless remedial actions are taken (Bennett, 1993; Griswold & Notkin, 1992).

Frazer (1992) identified some of the characteristics exhibited by the typical system viewed as a suitable candidate for reverse engineering:

1. Design specs are missing or incomplete.
2. The code is poorly structured.
3. The system requires excessive corrective maintenance.
4. The documentation is out of date.
5. Some modules have become overly complex.
6. Migration to a new software platform is required.
7. Migration to a new generation of hardware is required.
8. Hard coded parameters are subject to change. (p. 217)

Jacobson and Lindström (1991) noted all systems have a limited lifetime, independent of application domain or technological base. Each change to a system erodes the structure, making the following change more expensive. Eventually the cost of changes will become too high and the system will not be able to support its function.

Corbi (1989) said as changes and enhancements are introduced into maturing systems structure begins to deteriorate; design is altered by modifications; data structures are altered; documentation becomes outdated; key systems become less and less maintainable.

Business Changes

Welch and Grumman (1993) said most existing data processing systems were originally designed to do a single specific job. Systems were not designed in anticipation of changes in the way an organization does business. As functions are changed, added or extended the application eventually is incapable of supporting them.

Business Process Reengineering

Some of the current popularity of reverse engineering is driven by the interest in business process reengineering. Davenport (1993) said in the face of intense competition and business pressures of the 1990s, businesses must achieve 50 to 100 percent improvement levels in key processes. This interest in process improvement or business process reengineering (BPR) requires a basic reorganization of the business processes that underlie existing information systems. Many managers are beginning to realize that information technology applied to "broken" processes is not an effective use of resources. Ulrich

(1991) estimated productivity gains of more than 70 percent are possible if companies examine the processes currently supporting their business and redesign them to reflect efficient ways to achieve organizational objectives.

Client/Server Technology

Currently, client/server architecture is an area of great interest. According to Turner, Neuse and Goldgar (1993), many factors are driving the trend toward client/server processing: users are demanding easier, faster access to information and applications; information system budgets are being reduced in terms of the overall revenue percentage; and the capacity and capability of smaller machines and networks has improved. They believe the shift away from monolithic mainframe environments requires organizations to understand their legacy systems: A fundamental question for migrating a legacy system to the client/server environment is what part will run on the server and what part will run on the client? Software understanding is required to answer these questions.

According to Hayes (1994), successful recovery and regeneration requires a legacy application with a well-designed architecture, a rationalized data model, and a high degree of structure in its processes. Older, unstructured applications contain too many convoluted and redundant data structures and procedures to provide a useful base for reverse engineering

Object-Oriented Technology

According to Keyes (1992), object-oriented techniques are seen as a route to enhanced information systems productivity. There is great interest in this area because of the promises being made by proponents of the techniques. From a reverse engineering perspective, it is not clear what the relationship should be with object-oriented analysis, design, and programming. From a practical standpoint, it may be that reverse engineering is independent of eventual target implementation, particularly in the case of functional design recovery to support system replacement.

Software Maintenance

The ever-increasing cost and complexity of legacy system maintenance is one of the major drivers for the interest in reverse engineering. Friedlander and Toothman (1994) suggested that less than 10 percent of any information system budget is being directed at competitive advantage because the demand for system maintenance consumes more than 50 percent of professional resources in most organizations.

Jones (1986, as cited in Corbi, 1989) said the major difference between new development and enhancement work is the enormous impact that the base (existing) system has on key activities. As an example, in a new system design, user's requirements are explored and then moved into design; in an enhancement project, the user's requirements are often forced to fit into existing data and structural constraints. A significant portion of the design effort is therefore devoted to exploring the current programs to determine how new features can be added, as well as their impact on existing functions.

Arango, et al. (1985) noted the impact on maintenance of missing authors. In most cases, the software maintainers are not the original authors, are usually distant in time from the original implementation, and are likely to regenerate approximations of the original abstractions that were used. Avoiding approximation is difficult, and approximation errors are typically amplified by repeated maintenance steps. Over its lifetime, a system is modified until it bears little resemblance to its original structure.

Griswold and Notkin (1992) identified another maintenance problem: maintaining structure is a complex and costly activity because two logically independent software activities--maintenance (correction, enhancement, retargeting) and restructuring--are intermingled in almost all software process models.

Reverse Engineering Economics

As FIPS Pub 106 (1984) advises government information systems managers, there comes a time when all information systems must be redesigned. A major concern is how to determine whether a system is hopelessly flawed or whether it can be successfully maintained.

Sneed (1984) described the results of an effort to reengineer one large system. Respecifying the application programs took 17 man-months to complete--about one person-month of specification per 1400 lines of code. The ratio of program code to

specification documentation averaged 3 to 1 (i.e., for three pages of code there was one page of specification documentation). Discussions with users identified problems with the recovered specifications; four man-months was required to revise the specifications to accommodate the user's views. Sneed estimated the cost of the reengineering effort was two-thirds of the original development cost.

Sakthivels (1993) identified two major costs associated with maintenance. Deterioration cost is the increase in the maintenance cost. Obsolescence cost is the savings foregone by not using the latest technological developments to reduce maintenance costs. This cost also includes the loss of revenue by not using the improved substitute.

Jacobson and Lindström (1991) developed a matrix based on changeability and business value to aid in making decisions about old software (see Figure 7).

Ulrich (1991) cited a similar software option strategy matrix developed by PRISM and Hammer based on the organizational impact and the functional condition of a system (see Figure 8).

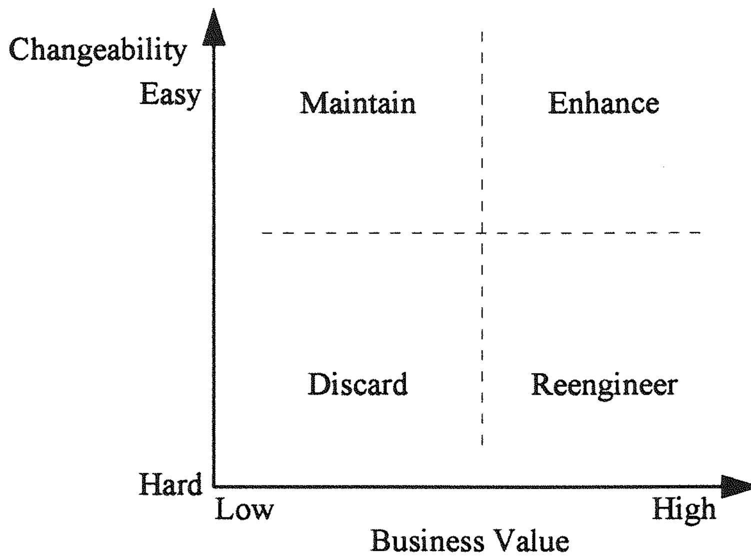


Figure 7. Decision matrix: what to do with an old system.

Note. Adapted from "Re-engineering of Old Systems to an Object-Oriented Architecture," by I. Jacobson and F. Lindström, 1991, *Proceedings of OOPSLA 1991* (p. 341). New York: Association for Computing Machinery.

Functional Condition	Organizational Impact: Low	High	Low	High
Good	Maintain or Technology Migration	Technology Migration	Maintain	Maintain
Poor	Phase Out	Replace	Maintain or Enhance	Enhance

Figure 8. Software option strategy matrix.

Note. Adapted from "Business Re-engineering and Software Re-engineering: The Relationship and Impact," by W. M. Ulrich, 1991, *CASE Trends, 3,* p. 36.

Reverse Engineering

Literature on reverse engineering is not extensive, although the amount of published information is increasing. Available information is frequently associated with software reengineering and maintenance. The connection between reverse engineering and maintenance is far from coincidental. As Cross, et al. (1992) noted, software reverse engineering is tightly coupled with software maintenance because maintenance activities have provided the motivation for many reverse engineering tools.

Rekoff (1985) defined reverse engineering as the "act of creating a set of specifications for a piece of hardware by someone other than the original designers, primarily based upon analyzing and dimensioning a specimen or collection of specimens" (p. 244). Rekoff's definition is concerned with hardware (of any kind) and reflects the origin of software reverse engineering in other engineering fields.

Reverse engineering was originally conceived to support software maintenance and was developed in that area (Canfora, Cimitile & Munro, 1994). According to Garnett and Mariani (1990) reverse engineering "involves the reversal of the design process . . . to restructure or document the code" (p. 186).

Rochester and Douglas (1991) proposed a reverse engineering definition that stresses its relationship to reengineering: reverse engineering recaptures the essential design, structure, and content of a complex computer system. Reengineering restructures a

system to take advantage of new technology without changing functions and features. The two processes are closely related because there is no systems reengineering without first reverse engineering their content.

Ulrich (1990b) referenced the IBM User Group Guide for a definition of reverse engineering: "The process of extracting, standardizing and documenting data descriptions and program logic from an implementation-dependent form to an implementation-independent form and migrating to an automated software engineering environment" (p. 42). This definition leans toward the view that reverse engineering is a part of the larger reengineering process.

Breuer and Lano (1991) made a distinction between reverse engineering and inverse engineering. They said reverse engineering is going all the way back to the design stage from the source code, while inverse engineering is going back only as far as the specification.

Benedusi, Cimitile, and de Carlini (1992) described reverse engineering as a collection of theories, methodologies, and techniques to support: (a) the design and implementation of a process to extract and abstract information from existing software and the production of documents consistent with the code, and (b) the addition of knowledge and experience that cannot be automatically reconstructed from code to these documents.

Tilley, et al. (1993) said reverse engineering is the identification of a system's current components and their dependencies, and the extraction of system abstractions and design information.

O'Hare and Troan (1994) described "incremental reverse engineering" as the ability to process different modules of a software system at different times (as opposed to all modules at the same time).

Cross, et al. (1992) noted that the continuing evolution of large, long-lived systems leads to lost design information. Reverse engineering, particularly design recovery, is a way to salvage whatever is possible from the existing system.

Sneed (1992) viewed reverse engineering as a process of deriving a specification from the original program source code with less emphasis on automation and more on supporting the human software engineer.

Bennett (1993) observed that the need for reverse engineering can arise for many different reasons, and there are many different ways of performing reverse engineering (including functionality changes).

Chikofsky and Cross (1990) maintained reverse engineering can be performed at any level of abstraction and at any stage of the life cycle because it does not involve changing the

subject system or creating a new system based on the reverse-engineered subject system. Reverse engineering is thus viewed as a process of examination, not a process of change or replication. Chikofsky and Cross identified two subareas of reverse engineering, redocumentation and design recovery:

1. Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views (e.g., data flow, data structure, and control flow) intended for a human audience.
2. Design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to observations of the subject system to identify meaningful higher-level abstractions beyond those obtained directly by examining the system.

Karakostas (1992) offered a more formal definition of reverse engineering that stresses the transformation from language X to language Y, where Y is a form more understandable to humans. Karakostas claimed it is often desirable to reverse engineer a system to a user oriented domain model (i.e., a conceptual model or a requirements model). This kind of reverse engineering is based on three kinds of knowledge: (a) knowledge about the software model (source code), (b) knowledge about the application domain, and (c) knowledge about transforming the software model to the domain model.

Early versions of structured analysis and design techniques suggested the first step in developing an information system should be to prepare a model of the current system

(Yourdon, 1989a). Building the current system model often resulted in a great deal of time being spent on a problem that was difficult, if not impossible, to resolve. Current structured methodology (Yourdon, 1989a) suggests a current model of the existing system is not built unless it is absolutely necessary. This step was dropped because a good methodology for extracting design information from legacy systems does not exist.

Munro (1992) identified four levels of reverse engineering: (a) inverse (step back to engineering specification), (b) renovation (step back to design), (c) reengineering (step back to code), and (d) redocumentation. Redocumentation is included as a reverse engineering technique because it allows some degree of overall system understanding without being concerned about how the program works.

According to Choi and Scacchi (1990), reverse engineering is used to first generate a design description from an implementation description, then to generate a specification description from the design description. It requires abstraction of four system properties:

1. Structural - Described by the resources exchanged among modules and subsystems through interconnected interfaces.
2. Functional - Described by the semantics of the exchanged resources. For example, operational resources (those that perform an operation) are abstracted by precondition and postcondition assertions. Non-operational resources (those that store a value) are abstracted by type definitions.

3. Dynamic - Described by the procedural algorithms that transform imported resources into exported resources. Dynamic properties are intramodular.
4. Behavioral - Described by the behavior of its objects (modules) in terms of relations among objects, their attributes, and the actions that manipulate them.

Harandi and Ning (1990) identified backward program abstraction steps and their related forward program development steps:

1. Implementation Level - Abstracts a program's language and implementation-specific features. Requires knowledge of language syntax and semantics, and possibly some knowledge of the implementation representation.
2. Structure Level - Reveals structure from different perspectives; results in an explicit representation of the dependencies among program components.
3. Function Level - Relates pieces of a program to their functions to reveal logical (as opposed to syntactical or structural) relationships.
4. Domain Level - Replaces the algorithmic nature of the function level with concepts specific to the application domain. For example, in the context of student record keeping, a program functionally understood as 'computing average by summing its inputs divided by the number of inputs' is interpreted as a 'grade-point-average computation' routine.

Darlison and Sabanis (1993) suggested reverse engineering is concerned with creating models of existing systems, in much the same way as 'normal' system specifications are

concerned with making models of non-existent systems. Reverse engineering is more or less synonymous with *system understanding*. According to Cross, et al. (1992), structural analysis of source code can result in code understanding in and of itself, "however, if humans do not ascribe meaning to code structures, structural analysis cannot determine the function of the code, neither in isolation nor within a larger organizational framework" (p. 220). If this statement is accepted as being true, it follows that computer-based function recovery from code is not possible. Cross, et al. also addressed this issue. They argued that reverse engineering tools *facilitate* the generation or regeneration of graphical program representations (e.g., data flow diagrams, control flow diagrams, structure charts, and entity-relationship diagrams) from other forms. Non-graphical representations can also be created to form an important part of system documentation. The significant point is that these representations do not present information that is not already contained in the program source code; they merely portray it in a different manner.

Holloway (1992) proposed that reverse engineering process be viewed in terms of moving through four distinct stages:

1. Stage 1 - Reverse Construction. Involves turning code into program design, JCL into job descriptions, and database schema to physical database design structures.
2. Stage 2 - Reverse Internal Design. Involves the translation of program design and job descriptions into dialogue design, batch suite screens, and screen and report designs.
3. Stage 3 - Reverse External Design. Involves the translation of dialogue design, batch suite design, and transaction network design.

4. Stage 4 - Reverse Detailed Requirements. Involves the translation of physical database design into a conceptual data model, and the translation of transaction network design to a functional model.

Connal and Burns (1993) suggested a four-step reverse engineering process: (a) constrain the system, (b) organize the components and data structures, (c) identify and rectify terminology redundancies, and (d) develop current working documentation. The four steps are defined as follows:

1. Constrain the System - In conjunction with discussions with users, analyze JCL or link maps to determine the scope of the system.
2. Organize the Components - All system components must be brought together and organized into a single repository for control and maintenance.
3. Document the System - The current system must be documented by mapping external linkages and data element flows through the system.
4. Identify and Rectify Terminology Redundancies - Legacy systems contain the same or very similar data names that refer to completely different business terms.

Ulrich (1990b) noted progress in reverse engineering has been made in two key areas: repository technology and data reverse engineering. Process (functional) reverse engineering efforts have not been as successful. As Breuer and Lano (1991) observed, many commercial software packages generate documentation and information about data

structure and program control flow from source code, but are not capable of identifying the functionality of the code.

Walker (1994) claimed the publicity related to technology success in general and of computing in particular gives people the impression that all problems can be solved by technology if enough effort is applied. The focus on successful efforts ignores attempts that end in failure; the publicity given to "automated reverse engineering tools" falls into this category.

Rekoff (1985) eloquently summarized the difficulty associated with reverse engineering:

> It should be recognized that the business of reverse engineering is not really greatly different from that of detective work in a criminal investigation or of conducting military intelligence operations. One has a cornucopia of what seems to be trivial and unrelated information that must be glued together in such a way that it provides the information required to resolve the need. (p. 245)

Reverse Engineering Objectives

According to van Zuylen (1993), understanding is one of the main objectives of reverse engineering. Chikofsky and Cross (1990) said the primary purpose of reverse engineering is to increase the overall comprehensibility of a system for both maintenance and new development.

Munro (1992) argued that an objective of inverse (reverse) engineering is to use formal transformation to achieve intellectual system understanding. Formal in this case means using logical representations of systems that can be mathematically manipulated.

Debest, Rüdiger, and Wagner (1992) suggested that the objective of reverse engineering is to recover something that would not have been lost if quality standards had been followed throughout the software development, operation, and maintenance process.

Frazer (1992) argued that the primary purpose of reverse engineering is to aid in system comprehension and to provide a basis for maintenance or future development. Frazer identified six reverse engineering objectives: (a) facilitate reuse, (b) provide missing or alternative documentation, (c) recover lost information, (d) assist with maintenance, (e) migrate from one hardware or software platform to another, and (f) bring the system under control of a CASE environment.

The Basis for Reverse Engineering

Reverse engineering is based on five fundamental theories (Chen, et al., 1990):

1. Theory One - Explicit representations of structural and functional code elements will aid program understanding.
2. Theory Two - Representations can be classified as either structural or functional. The structural view identifies the components making up the software. The functional view describes the application's functionality and subfunctionality.

3. Theory Three - The structural model consists of three views: part-of, connected-to and path.
4. Theory Four - A role or functionality can be associated with each element of the structural view.
5. Theory Five - The functional hierarchy associated with the program is important during maintenance. This functionality is not related to requirements specification and design but to the dynamic characteristics of an application.

Reverse Engineering Problems

Chikofsky and Cross (1990) noted the term "reverse engineering" originated from the analysis of hardware. Reverse engineering is regularly applied to identify hardware designs from finished products. The hardware objective is to duplicate the item. The software objective (ignoring illegal reverse engineering activity performed with the intent of producing a similar product) is most often to gain a sufficient design-level understanding to aid maintenance, strengthen enhancement, or support replacement.

Program understanding has been compared to natural language understanding (DeBaud, Moopen, & Rugabers, 1994). Most current reverse engineering techniques are based on program structure analysis using lexical, syntactic, and semantic rules because these techniques are well known. However, program understanding based on structure alone is as difficult as understanding essays, articles, or stories based solely on knowledge of rules of English grammar.

McCabe and Williamson (1992) believe that reverse engineering exists to support forward engineering. Additionally, they implied the results of the reverse engineering process can be ported to a CASE tool to support forward engineering. This article, which appeared in a trade magazine, did not adequately explain how the porting could be performed.

Pfrenzinger (1992) made a similar claim when he said the purpose of reverse engineering is subsequent enhancement or replacement via forward engineering. Pfrenzinger said reverse engineering can automate the manual step of understanding a system prior to changing or replacing it. This article also appeared in a trade magazine and did not offer any information about how this understanding could be achieved.

Darlison and Sabanis (1993) argued that it can be shown mathematically that some information cannot be derived automatically from source code because of the undecidable nature of the associated mathematical problem.

Grumman and Welch (1992) argued that it is not possible to extract from application code a formal, functional, nor technical specification of an application. Grumman and Welch stated that, in general, it is possible to say only what designers and programmers did, not what they wished to do.

Rochester and Douglas (1991) suggested that, although reverse engineering is reasonably obvious in concept, the layers on layers of old, maintained code written in a variety of languages characteristic of legacy systems makes it highly complex technically.

According to Kozaczynski, et al. (1992) reverse engineering requires that programming concepts (e.g., instructions, variables, control structures) be recognized and associated with generic data objects and algorithms. The meaning of these objects must then be described in problem domain terms. Identified concepts, however, may have no donain equivalent; when they implement platform-specific technical tricks for example.

Abstracting concepts in the application domain implies the use of informal knowledge external to the software system and necessitates human intervention because some of the information essential to the task is not present in source code and documentation (Bachman, 1988; Canfora, Sansone, & Visaggio, 1992).

Arango, et al. (1986) asserted that human experience in reverse engineering is vital. It is necessary to rely on a maintainer's experience and knowledge of the application domain as well as on available documentation.

Antonini, Benedusi, Cantone, and Cimitile (1987) identified a problem frequently encountered in reverse engineering (see Figure 9). Design components (A) may not be

found in code (B). Code components (C) may have no equivalents in design. B is the area of consistency between between design and code.

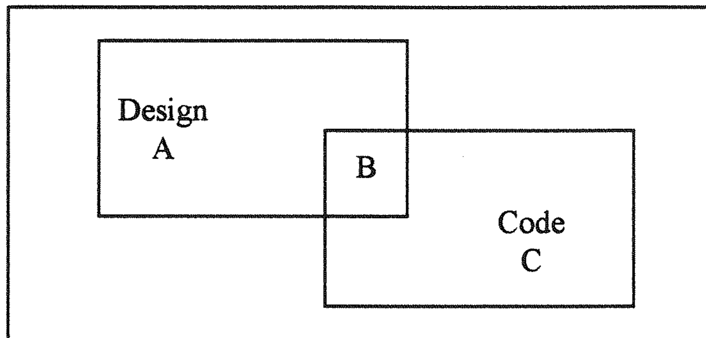


Figure 9. Virtual overlapping between code and program design.

Note. Adapted from "Maintenance and Reverse Engineering: Low-level Design Documents Production and Improvement," by P. Antonini, P. Benedusi, G. Cantone, and A. Cimitile, 1987, *Proceedings of the IEEE Conference on Software Maintenance* (p. 91). Los Alamitos, CA: IEEE Computer Society Press.

Grumman and Welch (1992) maintained it is not possible to extract the functional specification from the application code, but it is possible to document the functionality to support decisions about whether, how, and at what cost the application can be overhauled.

Canfora, et al. (1994) proposed that the effort required to produce a descriptive specification is generally less than the effort required to produce an operational specification. The reverse engineering process first sets up low-level design documents to aid in understanding the functions the software implements, and then tries to reach the specification level by means of successive abstractions.

Wilde, Gomez, Gust, and Strasburg (1992) observed that although software engineering practice dictates saving the mappings from user functionalities to code segments, it is relatively rare to encounter a project that still conserves these mappings after a prolonged period of maintenance. Even if traceability was provided during development, this documentation is often the first casualty of the time pressure associated with keeping a system operational.

Warden (1992) identified another major problem associated with reverse engineering. During system specification and design a significant amount of non-procedural business knowledge is used to make system architecture, data design, and procedural processing decisions, but these decisions are seldom documented, maintained, and made available during maintenance.

Warden (1992) divided reverse engineering into a family of tasks at three major levels:

1. Implementation Level - Concerned with documenting code characteristics such as program structure, control flow complexity, internal data complexity, and standards violations.
2. Design Level - Concerned with documenting design characteristics such as modularity, coupling, cohesion, depth factoring, and file design complexity. May be documented at a partial or global design level.
3. Business Level - Concerned with documenting in a nonprocedural way the business functions which a system performs. The descriptions obtained are design independent.

Byrne (1991) suggested the most important problem in reverse engineering is implementation bias. It is necessary to separate design information from implementation information. For traceability, the recovered design should record links between recovered design and the original sources. Byrne also concluded that domain information can aid in recovering information about the purpose and significance of a function.

Pfrenzinger (1992) indicated it is much easier to determine "how" an existing system operates, than it is to determine "what" and "why" it operates. Code does not contain the information to determine the "what" and "why." If it does, it is often so obscure that it would require an expert to decipher the code or add the missing information. One of Pfrenzinger's main points is that the higher the target level on the reverse engineering scale, the less automatic and the more manual the reverse engineering process becomes.

Frazer (1992) identified interfaces with other systems as a potential problem area in reverse engineering because it is difficult to abstract interface design information from only one side of the interface. Frazer said considerable manual effort is required to understand interfaces.

Hickey and Jennings (1994) observed that programs are not capable of understanding a business, reading code, and making a connection between the two. The essential elements of system design can only be developed by people who understand the business problem

and who are experienced with the internal detail of the existing system. In addition, it takes human beings to read the code, infer its meaning, and recast it in a structured form.

Design Recovery/Inverse Engineering

Robson, et al. (1991) defined inverse engineering as the process of extracting high-level representations from source code. Inverse engineering involves screening out noise present in source code to provide a more abstract view of a system.

Wilde, et al. (1992) claimed locating user functionalities in existing system source code is a special case of the general problem of design recovery. They reported that, although many sophisticated methods for design recovery have been proposed, all of the work involved static rather than dynamic analysis. They suggested the best sources of information for design recovery, if available, are the developers and maintainers who have experience with the system.

Biggerstaff (1989) agreed, saying source code does not contain much original design information. Biggerstaff said additional information sources, both human and automated are necessary. Design abstractions must be developed from a combination of code, existing documentation, personal experience, and general knowledge about a problem and the application domain.

Lenihan (1993) saw design recovery as the fifth and final phase of a refurbishment effort: "Design recovery captures certain elements of the current system design, incorporates these elements into a Computer Aided Software Engineering (CASE) tool and provides engineers with the ability to accurately document the functional and technical aspects of the system" (p. 23).

Existing Reverse Engineering Procedures

While useful data reverse engineering tools are available, process reverse engineering tools are not. Artificial intelligence and knowledge-based systems have been the subject of considerable research, but have not been implemented in commercial tools.

Table 4 is a comprehensive, but not exhaustive, list of 59 research and commercial reverse engineering tools, techniques, and methodologies developed between 1980 and 1994. The table is arranged in chronological order and indicates the language or languages each tool accommodates. The comments column describes the general nature of the tool. Twenty-five tools (42 percent) are designed for COBOL or are language independent.

Eleven tools (19 percent) fall into the software physical structure category (i.e., control graphs, call graphs, structure charts, and syntax trees). Tools in this category are: Tool AURUM, IA, AdaAN, RETA, BAL/SRW, NuMIL, DPUTE, Schematics, MAP, Rigi, and UIFG. Six are suitable for use with COBOL.

Table 4
Reverse Engineering Tools and Methodologies

Year	Tool/Method name	Language(s)	Researcher(s)	Comments
1980	Tool AURUM	COBOL, et al.	Wagner	Visualization of software structure
1980	PUDSY	PASCAL	Lukey	Program schemata matching
1982	Eureka Countdown	Langugage independent	Zvegintzov	Physical program inspection
1983	MAP	COBOL	Warren	Paragraph structure charts
1985	ME2	PASCAL	Collofello & Blaylock	Syntactic analyzer for maintenance
1985	PROUST	PASCAL	Johnson & Soloway	Knowledge-based program understanding
1986	TMM	Common LISP	Arango, Baxter, Freeman, & Pidgeon	Transformation
1987	IA	COBOL	Antonini, Benedusi, Cantone, & Cimitile	Control flow graphs, nested trees, cross references
1988	Programmer's Apprentice	Ada	Rich & Waters	Program language learning tool based on program plans
1988	NoName-1	C	Calliss, Khalil, Munro, & Ward	Knowledge-based transformation to known plan
1988	LogiScope	COBOL, et al.	Meekel & Viala	Commercial tool - control and call graphs, complexity analysis
1988	MicroScope	Common LISP	Ambras & O'Day	Knowledge-based using frames and rules
1988	PAT	PASCAL	Harandi & Ning	Knowledge-based cliché recognition
1989	AdaAN	Ada subset	Gopal & Schach	Visibiltiy flow graphs
1989	PUNS	Assembler	Cleveland	Program information database
1989	DESIRE	C	Biggerstaff	Variation of program plans for program understanding
1990	PM	Ada, et al.	Reynolds, Maletic, & Porvin	Knowledge-based program understanding
1990	STREAM	Amore, PROLOG	Karakostas	Domain modeling

Table 4. (continued)

Year	Tool/Method name	Language(s)	Researcher(s)	Comments
1990	RETA	Assembler	Chen, Heisler, Tsai, Chen, & Leung	Program syntax tree representation
1990	BAL/SRW	Assember	Kozaczynski	Program structure charts
1990	Alchemist	C	Garnett & Mariani	Software reclamation for reuse
1990	NuMIL	C	Choi & Scacchi	Program structure recovery
1990	CSS	COBOL, FORTRAN	Breuer & Lano	Program transformation
1990	Recognizer	Common LISP	Rich & Wills	Program cliché recognizer, graph parser
1990	NoName-2	Language independent	Hausler, Pleszkock, Liner, & Hevner	Function abstraction
1990	REFINE	Language independent	Burson, Kotik, & Markosian	Database-based transformation; program templates
1991	ReForm	Assembler	Bennett	Transformation
1991	Maintainer's Assistant	Assembler	Yang	Program transformation to Z
1991	DPUTE	COBOL	Joiner, Tsai, Chen, Subramanian, Sun, & Gandamaneni	Modified COBOL dependence graphs (data centered)
1991	COBOL/SRF	COBOL	Kozaczynski, Letovsky, & Ning	Knowledge-based program understanding.
1991	SEES	COBOL, C	Avellis, Iacobbe, Palmisano, Semeraro, & Tinelli	Knowledge-based assistant
1991	Source/RF	COBOL, JCL	Napier	Commercial tool
1991	Schematics	Language independent	Lerner	Program structure graphic
1991	LaSSIE	Language independent	Devanbu, Brachman, Selfridge, & Ballard	Knowledge-based

Table 4. (continued)

Year	Tool/Method name	Language(s)	Researcher(s)	Comments
1992	IASSys	Ada	Canfora, Sansone, & Visaggio	Dynamic data flow diagrams
1992	FACET	COBOL	Howden & Pak	Structural and logical abstractions
1992	TRANS	COBOL	Kozaczynski, Ning, & Engberts	Knowledge-based program plans and transformation
1992	COBOL/Analyst	COBOL	Eliot	Commercial tool
1992	IRENE	COBOL	Karakostas	Domain knowledge-based
1992	NoName-3	COBOL, C	Grumman & Welch	Functional, directed graphs
1992	Rigi	COBOL, et al.	Müller, Tilley, Orgun, Corrie, & Madhavji	Subsystem composition graphs
1992	DMS	Language independent	Baxter	Design maintenance system
1992	NoName-4	PASCAL	Benedusi, Cimitile, & de Carlini	Hierarchical data flow diagrams
1992	Data_tool	PASCAL, PROLOG	Canfora, Cimitile, & de Carlini	Knowledge-based intermodular data flows
1993	Web structures	ALGOL-60 subset	Maggiolo-Schettini, Napoli, & Tortora	Transformation
1993	WSL	Assembler	Ward	Program transformation
1993	UIFG	C	Harrold & Malloy	Unified interprocedural flow graphs
1993	RECAST	C	Edwards & Munro	Convert source code to SSADM
1993	Via/Renaissance	COBOL	Lanubile & Visaggio	Commercial tool
1993	REDO	COBOL	Lano, Breuer, & Haughton	Program transformation to Z
1993	Legacy Workbench	COBOL	Hayes	Commercial tool
1993	ARM	Language independent	Keller & Nance	Abstraction refinement

Table 4. (continued)

Year	Tool/Method name	Language(s)	Researcher(s)	Comments
1993	MGAP	PASCAL	Laffick	Modified goal and plan language learning
1994	PIAS	C	Khan	Adiabatic multi-perspective abstraction
1994	RE-Analyzer	C	O'Hare & Troan	Data flow diagrams, entity-relationship diagrams
1994	NoName-5	C	Quilici	Program plans recognition (theoretical)
1994	QDA	CMS2, Assembler	Howden & Wieand	Informal correctness checking
1994	Episodic Processes	None	Von Mayrhauser & Vans	Program comprehension process
1994	SeeSYS	Proprietary	Baker & Eick	Large system visualization

Nine tools (15 percent) are classified as knowledge-based program understanding tools. Tools in this category are: PROUST, NoName-1, MicroScope, PM, COBOL/SRF, SEES, LaSSIE, IRENE, and Data_Tool. Four are suitable for use with COBOL.

Eight tools (14 percent) fall into the transformation category. Tools in this category are: TMM, CSS, REFINE, ReForm, Maintainer's Assistant, Web Structures, WSL, and REDO. Three are suitable for use with COBOL.

There are eight tools (14 percent) in the program plans category. This category includes program plans, program clichés, program schema, and program templates. Techniques in this category may overlap other categories. For example, a tool locates an unknown program plan in source code and matches it with an existing plan in a plan library; after the match, the unknown plan is replaced by a known plan. While this is actually a form of transformation, the underlying principle is the program plan. Tools in this category are: PUDSY, Programmer's Apprentice, PAT, DESIRE, Recognizer, TRANS, MGAP, and NoName-5. Only one tool is designed for COBOL.

Four tools (7 percent) fall into the data flow diagramming category. Tools in this category are: IASSys, NoName-4, RECAST, and RE-Analyzer. None are designed for COBOL.

There are three tools (5 percent) in the functional abstraction category. The focus in this category is abstraction--moving away from source code to a higher level of knowledge.

Tools in the category are: NoName-2, FACET, and ARM. All are oriented for use with COBOL.

Five tools (7 percent) are commercial products. Included in this group are: LogiScope, Source/RF, COBOL/Analyst, Via/Renaissance, and Legacy Workbench. All are suitable for use with COBOL.

Eleven tools (19 percent) do not fit into any of the other major groups. The Eureka Countdown is one of the few techniques based on manual code examination. PUNS is a support tool based on the construction and automatic population of a program information database. PIAS (adiabatic multi-perspective abstraction) takes a revolutionary approach to reverse engineering. Episodic Process is an explanation of the program comprehension process rather than a tool. Other tools in the category include: ME2, STREAM, Alchemist, NoName-3, DMS, QDA, and SeeSYS. Three tools support COBOL.

Software Physical Structure

Physical structure representations do not contribute significantly to reverse engineering. Reverse engineering focuses on recovering high-level functional design information from source code. Tools concentrating on code-level information are more suited to program maintenance than to reverse engineering; most commercial tools, except data structure recovery tools, fall into this category.

Knowledge-based Program Understanding Tools

Knowledge-based program understanding tools apply artificial intelligence techniques (e.g., expert systems and predicate logic) to support software understanding. Mathematical or logical models are frequently used to represent programs. The fundamental concept in this approach is that all properties of a program can, *in principal*, be discovered from the text of the program itself by means of purely deductive reasoning--the application of valid rules of inference to sets of valid axioms. As Biggerstaff (1989) noted, research tools are applied to small-scale problems and are not focused on informal information sources.

Generally speaking, computer-based tools in this category attempt to model the way people understand programs or extract new information from source code by making inferences from existing information. Like other areas of artificial intelligence, the use of knowledge-based techniques in software reverse engineering research has not been extremely effective. Computer-based reverse engineering tools based on artificial intelligence have met with limited success, even with small programs. There appears to be little practical value for these tools in a real world environment. As Tan and Dietz (1994) noted, program understanding is essentially a human-centered activity, not a machine-centered activity.

Transformation Tools

Transformation tools automatically transform source code into more readable or understandable forms. Transformation tools focus on low-level program information and are generally more suited to reengineering than reverse engineering. Transformation tools frequently produce program representations (i.e., Z) that are more difficult for people to read and understand than the original source code. However, these representations are more easily processed by computers, and are often used to transform unstructured code to structured code, or to convert one language to another; they are seldom applied to raise the level of abstraction--the goal of reverse engineering.

Program Plans

The program plan approach to reverse engineering takes an unknown plan or structure and identifies it to a known plan. The collection of known plans then equals program understanding. This category also focuses on code-level knowledge, although there is a slight degree of abstraction away from pure programming language in some tools. Some transformation tools apply the program plan technique by substituting a plan in one language for the same plan in another language.

There are several problems associated with the plans approach to reverse engineering. One problem is the need for a large plans library against which source code can be compared. Another problem is that source code corresponding to a program plan may be dispersed in multiple parts of a program. A third problem is that search and compare

operations are severely impacted by combinatorial explosion, although some tools have implemented techniques to limit searches. This approach has only been successful with small programs containing simple logic and is not considered viable for practical reverse engineering.

Data Flow Diagrams

Traditional data flow diagrams (DFD), usually associated with requirements analysis, are an excellent way of graphically describing a network of external data sources and destinations, processes, and data stores connected by data flows. Each primitive (bottom-level) process has an associated process description to explain details of the process that cannot be shown graphically.

DFD based on program source code, however, portray physical details of program structure and operation in a graphical format rather than in a textual format. If traditional DFD (based on high-level functions) are generated from a reverse engineering process, some functional abstraction activity--possibly manual--would have been required to produce them. In this sense, functional abstraction DFD are a means of displaying reverse engineering results rather than actual reverse engineering.

Functional Abstraction Tools

Functional abstraction, the category with the fewest tools, is a step in the right direction for reverse engineering. However, these tools are still in the research stage. If they can be

developed at all, computer-based abstraction tools suitable for practical application are many years in the future.

Computer Assisted Reverse Engineering (CARE) Tools

The use of current computer-based tools in relation to reverse engineering offers little in the way of capturing functional information from legacy systems. Available reverse engineering tools are useful for automatically extracting database management system structure directly from COBOL data division entries.

Control graphs, call graphs, data flow graphs, structure charts, entity-relationship diagrams, logic flow diagrams, reserved word reports, and variable "where-used" reports are relatively easy to extract from source code. Commercial reverse engineering tools are typically capable of generating these products. However, these documents cannot capture and represent semantic abstractions as the functionality associated with software/data structure.

The U.S. Air Force Software Technology Support Center Re-engineering Tool Report (Sullenaur, Olsen, & Murdock, 1992) listed 67 products classified as reverse engineering tools. However, most of the tools are not reverse engineering tools. Of the 67 tools listed, only 11 are used with COBOL (see Table 5).

Table 5
Nominal Reverse Engineering Tools Available Commercially

Tool/Method name	Comments
Application Browser	Produces documentation
Autoflow	Produces flow charts and functional calling trees
ENVISION	Produces documentation
IMSCASE	Imports code to KnowledgeWare's ADW CASE Tool
InterCASE	Imports code to ADW Design Work Station
InterCycle	"Reverse engineers" code into a repository
Logiscope	Analyzes source code complexity
PM/SS	Performs impact analysis
REFINE/COBOL	Performs redocumentation and code conversion
REVENGG	Abstracts structure and program interaction
SOFTWARE Refinery	Performs redocumentation and code conversion

According to descriptions written by VIASoft, Incorporated (as reported in Sullenaur, et al., 1992):

> VIA/Insight is a COBOL analysis tool that *completely automates* [italics added] the understanding process for programmers. It captures and displays logic and data path information, giving programmers the data they need to understand and maintain existing programs.
>
> VIA/Renaissance is more of a truly reverse engineering product that provides for recovery and reuse of existing business applications that allows programmers to examine programs graphically or in source code form. (pp. B-110-111)

Neither of these products are classified as reverse engineering tools in another part of the same document.

According to the product literature for RE/Cycle (CGI, Berwyn, PA), the tool performs *semantic analysis* of applications, i. e.:

1. Data Division - Identifies relations between elementary data items and data structures; establishes copy books and files.
2. Procedure Division - Establishes relations between program entities.
3. Inter-program Analysis - Matches program to program to ensure components (file descriptions, inter-file relationships, and group-elementary item relationships used in linkage and common areas) are homogenous and create relationships.
4. Screen Analysis - Physical screen layout.
5. Data Standardization - Homonyms, synonyms, on-line edits, and updates.

This is an excellent example of the misuse of terms. Although the claim is that semantic analysis is performed, the examples given are syntactic analysis (i.e., the structure of the program is described rather than its meaning).

The final example of marketing material for a commercial tool is for Excelerator (Index Technology, Cambridge, MA). Their literature describes the capabilities of Excelerator for Design Recovery as including these features:

1. Reads COBOL source code, Information Management System (IMS) database definitions, IMS/Message Format Services (MFS), Customer Information Control System (CICS)/Basic Mapping Support (BMS); and generates physical models stored in a dictionary. IMS/MFS and CICS/BMS data is converted to screen designs.

2. Produces structure charts showing the hierarchy of paragraphs and sections as functions.
3. Produces data model diagrams from IMS database definitions; groups related fields into structures called segments; and shows the hierarchical relationships among the segments.
4. Produces data definitions by extracting the definitions [descriptions] from files, working storage, and linkage sections from the program data division, screen maps, and IMS segments; stores information in the dictionary.
5. Produces reports, including cross reference lists, where used lists, unreferenced paragraphs, unreferenced data, data item assignments, file input/output reports, and a measure of cyclomatic complexity.

The output of this tool, while primarily graphics based, is still at the program level. There is no design recovery; it is physical implementation recovery.

Except for potentially useful data design recovery, there are no commercially available tools that address the problem of design information recovery, despite claims to the contrary.

Summary

Early in the history of computing, machine costs were extremely high while personnel costs were low. The cost of programming systems was relatively small compared to the

high cost of computers. Huge computer systems were created by programming teams made up of many programmers. Analysis and design techniques were crude by today's standards, and systems were designed and developed without regard to future maintainability. It is not clear whether the anticipated life of a computer system was even a consideration when the bulk of original computer programming was occurring during the late 1960s and 1970s.

During the period of rapid original software development in the 1960s and 1970s, maintenance was a small part of the systems development life cycle. By the late 1970s, the activity required to maintain existing systems began to exceed the activity devoted to new systems development. By the 1980s, maintenance of existing software began to be recognized as a major problem for the information technology industry. It was evident old system architectures were constraining new designs.

By the 1990s, the effect of long-term maintenance of systems originally developed in the 1960s and 1970s was evident--each modification to an existing system increased the difficulty of the next modification. Systems that were not well engineered became maintenance nightmares under the brunt of numerous modifications and enhancements. Personnel costs for the maintenance and replacement of legacy systems became the single most expensive part of the software life cycle.

Documentation for legacy systems is frequently absent or outdated. In most cases, the only reliable source of information is the source code. Maintenance programmers are faced with the problem of not only trying to understand the intent of the original programmer, but also the intent of every maintenance programmer who has made a change to the system. For maintenance programmers, program understanding has become and increasingly important skill.

Program understanding is now a significant research subject, and many approaches have been proposed. One problem with this research, however, is that recovering design information from source code is more difficult than creating the software. Software psychology is one field of great interest in the information systems industry, but it is more descriptive than predictive in nature. Software psychologists are able to observe programmers as they write computer programs or try to understand existing programs, and they are able to describe the procedures followed. They are not able, however, to use this knowledge to appreciably reduce the complexity of software development and understanding. Computer software remains a unique activity not well understood.

Software reengineering techniques have been introduced to deal with the problem of maintaining legacy systems. Software reengineering focuses on extending the life of legacy systems by restructuring the code in accordance with modern software development techniques by rehosting applications from one computer platform to another, or by translating one language to another. Some success has been reported with

reengineering techniques, but it is not clear whether reengineering is less expensive than total systems replacement.

The "software is a form of mathematics" component of the information systems industry has offered proof that program transformation is logically possible and is a straightforward process that can be performed by a computer. Many computer-based reengineering tools (both experimental and commercial) have been developed. There is evidence these tools are useful in dealing with some aspects of the legacy system problem. There is also evidence, however, that the usefulness of these tools is often exaggerated. One aspect of reengineering tools seldom discussed is how much they can extend the life of legacy systems.

The inadequacy or the inappropriateness of reengineering has led to reverse engineering. The basic philosophy of reverse engineering recognizes that a legacy system must be replaced. The task of reverse engineering is to recover the business functions, business rules, and data structure contained in legacy systems and restate this information at an appropriate level of abstraction to support replacement. With the possible exception of creating documentation for legacy systems where none exists, reverse engineering is not considered in this dissertation to be a maintenance activity.

Data structure recovery from legacy systems is a relatively simple part of reverse engineering. Even if a legacy system is constructed around flat file structures, there are

reasonably straightforward procedures (both manual and automated) for performing data structure recovery (Keller, 1983). As data structure reverse engineering is less complicated than function or process reverse engineering, it is not directly addressed in this research. It is recognized, however, that a reverse engineering methodology would not be complete without techniques and procedures to capture data structure.

There are many reports in the literature on experimental reverse engineering methodologies and tools. A common characteristic of these methodologies and tools is that they are applied to relatively simple programs; it is often difficult to see how they can be applied to real-world systems consisting of millions of lines of code. In many cases, the resulting graphing techniques and alternative notations are more difficult to understand than the source code from which they were derived. In particular, logic-based approaches (artificial intelligence or expert system) are ineffective for application to large-scale systems.

The task faced by the reverse engineer is a difficult one. One point made clear by the literature is that complete design recovery from legacy system source code alone is not an achievable goal. Between system functional requirements and software program implementation, essential elements of information are lost. Although it might seem possible to apply the software development process in reverse, in effect undoing the forward engineering process, this missing information makes design information recovery extremely difficult.

Chapter III

Methodology

This chapter explains how the reverse engineering investigation was conducted.

The format of the investigation was centered around two methods: (a) basic research, and (b) exploratory development. Basic research is systematic, intensive study to gain knowledge and understanding of reverse engineering. Exploratory development is systematic application of reverse engineering knowledge to meet a specific need.

As discussed in Chapter I, establishing the reverse engineering methodology involved five phases: (a) approach selection, (b) methodology development, (c) case problem selection, (d) methodology application, and (e) methodology assessment. The last three phases are discussed in Chapter IV, Results.

Approach selection focused on identifying the basic reverse engineering methodology to be developed (i.e., knowledge-based, mathematical, abstraction). Approach selection was supported by a review and analysis of existing reverse engineering methods and tools, as well as a detailed analysis of the domain in which the methodology is to be employed.

Methodology development was based on a development plan centered around the information engineering method of performing requirements analysis. The development plan concentrated on the synthesis of procedures to produce the results identified as requirements.

Case problem selection involved the choice of a suitable application for evaluating the reverse engineering methodology. The major limiting factor in this phase was selecting an application small enough to be manageable, but large enough to be realistic.

Methodology application involved the use of the design information recovery technique on the selected case problem. The objective in this phase was to evaluate the methodology and to identify changes or enhancements to address problems encountered during the case study.

Research Methods Employed

The essence of reverse engineering is recovering information about a system design from the incarnation of the system--the program source code. Viewed from this perspective, reverse engineering can be considered an information processing problem. The input is known (legacy system source code) and the desired output is known (design information); what remains to be defined is the process to convert input to output. This process, although simplified, is the same faced by any information system developer.

However, it must be stressed that reverse engineering is a human information processing problem, and not only a computer information processing problem. Sufficient evidence in the literature supports the contention that design recovery from source code is an unsolvable problem for a computer system, and reasonable doubt exists as to whether a computer system alone will ever be able to extract design-related information from source code.

The information engineering approach to identifying and specifying requirements for an information processing system described by Miller (1995a, 1995b) formed the core for synthesizing the reverse engineering methodology. Application of this information engineering technique was modified slightly because the intent was to develop a manual reverse engineering methodology rather than a computer-based methodology. Computer implementation of the manual methodology is an independent problem and should be addressed in a separate study.

Specific Procedures Employed

Description of Phase 1 - Reverse Engineering Approach Selection

The objective of Phase 1 was to select a category or basic model of a reverse engineering methodology. There were five tasks in Phase I.

In Task 1.1, the problem was defined as clearly and precisely as possible. The operational environment (the application domain area) was described to establish the scope and boundaries of the investigation. The programming environment and the operational problems to be addressed by the methodology and the programming environment were also described.

In Task 1.2, a forward engineering reference model was developed. This model provided the framework for the construction of the reverse engineering methodology.

In Task 1.3, five distinct methodologies were selected from those described in Chapter II and analyzed in detail to identify applicable features, techniques, or methods for the specific problem application domain.

In Task 1.4, three program reference models were developed (batch, on-line, fourth generation language). These models portray the general structure of the various components found in the target system. The reference models were also used in Phase 2, reverse engineering methodology development.

In Task 1.5, the output products to be produced by the reverse engineering methodology were defined. The output products represent the final result of the methodology and describe the vehicle for presenting results.

Description of Phase 2 - Reverse Engineering Methodology Development

Phase 2 was the core of the research. The objective of Phase 2 was to describe a reverse engineering methodology suitable for use in the specific environment described in the problem definition. There are normally four tasks associated with this phase; the fourth task, prepare physical model, was omitted in this application. Each task consisted of a process component and a data component (defined later in this chapter). There were 3 tasks in Phase 2

In Task 2.1, the purpose, goals, and objectives of the reverse engineering methodology were identified, defined, and described. Functions of the methodology were described in a hierarchical form using key areas, tasks, sub-tasks and activities. The narrative description of functions was augmented by a visual process model. A high-level conceptual data model defining the data structure required to support the functions was produced.

In Task 2.2, the activities (primitive functions) from the conceptual model are normally expanded to add frequency, location, organization, and other information useful for implementing requirements in an information system; this activity was omitted in this application. The data model was expanded to include the "business rules" that define the relationships between the conceptual model entities, and a definition model was produced.

In Task 2.3, each of the conceptual processes was decomposed into a series of services. The services were linked in this step with pertinent data model components. The

conceptual data model was converted to a logical data model by applying table formation rules to each relationship. Attributes were assigned to the tables and the tables were normalized to the third normal form.

Description of Phase 3 - Case Study Subject Selection

The objective of Phase 3 was to select a subsystem or part of a subsystem to be used in the application of the reverse engineering methodology formulated in Phase 2. There were three tasks in this phase.

Task 3.1 involved establishing selection criteria for the case study subject. In Task 3.2, the method to be used to select two or more candidate components of the system was established. Task 3.3 was an evaluation of the candidate subsystems using criteria established in Task 1 and resulted in the identification of the final test case subject to be used in Phase 4.

Description of Phase 4 - Reverse Engineering Methodology Application

The objective of Phase 4 was to test the reverse engineering methodology developed in Phase 2 against the case study identified in Phase 3. There were three tasks in this phase.

Task 4.1 was the execution of the reverse engineering process model using the selected system component. The objective was to recover detailed design information from the source code. This was the "experimental" phase of the investigation. Task 4.2 was the

collection of analysis-related information. This included the time required to analyze various components, problems encountered during the analysis, and problem solutions. This information became the raw data for the investigation results described in Chapter IV. Task 4.3 was the analysis of statistical data generated during application of the methodology.

Description of Phase 5 - Methodology Assessment

The objective of Phase 5 was to assess the reverse engineering methodology in both qualitative and quantitative terms with respect to its usefulness in recovering design information from the specific application domain. The results of this phase are presented in Chapters IV and V. There were six distinct tasks in this phase.

In Task 5.1, the design of the system component was compared with reverse engineered design information. In Task 5.2, significant design discrepancies were identified in the reverse engineered model. Task 5.3 involved the analysis of differences between the original and the reverse engineered designs. In Task 5.4, design differences were evaluated. In Task 5.5, methodology faults were identified and assessed. The assessment was based on the design discrepancies identified in Task 3 and Task 4. In Task 5.6, possible methodology changes were suggested.

Execute the Reverse Engineering Synthesis Plan

This section describes the execution of the research methodology outlined in the previous section. The reverse engineering knowledge accumulated through the literature review

was assimilated and combined with existing problems and needs familiarity to formulate a structured reverse engineering approach. A high-level description of the procedures to be followed in recovering design information from source code and related documents was produced. A visual model of the methodology was presented in the form of leveled data flow diagrams. The conceptual, definition, and logical data structure models required to support the methodology were also produced.

Problem Definition

The problem addressed by this research is how to recover sufficient design information from an existing legacy system to support system replacement. A secondary problem addressed is how to recover source code information to generate high-level documentation when essential information is not available or is so outdated it is unusable. The software maintenance problem is implicitly addressed because of the close association between program understanding for design information recovery and program understanding for correcting and modifying legacy systems.

The Operational Environment

The system domain for the reverse engineering methodology is military logistics. The specific domain is logistics systems managed by the U.S. Air Force Logistics Command at Wright-Patterson Air Force Base, Dayton, Ohio. One system was selected as the subject for this reverse engineering investigation as a matter of convenience and accessibility, and because the researcher has experience with the system. Source code for an operational

system was provided by the Air Force with the understanding that functional users and system maintainers would not be able to support the investigation. In a real world setting, the reverse engineer would depend on functional users, maintainers, and developers for external system information.

Operational Problems

One of the fundamental logistics systems within the Air Force Logistics Command is a requisition processing system, the Stock Control and Distribution (SC&D) System. This system processes requests for items from Air Force bases and other agencies and tracks the issuance, financial accounting, and transportation of items from the issuing warehouse to the requesting unit. The SC&D system has existed in some form for many years and traces its ancestry to the second generation IBM 7080/7090 system.

In the mid-1980s, the SC&D System was modernized by converting it to an on-line system using a database management system. Software reengineering was the primary methodology used. Many batch programs were simply converted to operate as subroutines in an on-line mode. The large program sizes do not suggest there were extensive efforts to modularize the system nor to redesign it for easier maintainability.

The modernized system is now ten years old and has been subjected to intensive maintenance. Degradation due to maintenance has occurred and will eventually dictate the development of a replacement system.

A problem for the government when the next modernization program begins is that system redesign and maintenance knowledge may not be available. The contractor who performed the conversion and maintenance work will turn the system over to the government or possibly another contractor for continued operation and maintenance; the institutional knowledge regarding the system redevelopment will go with the contractor's employees.

COBOL Program Environment

The SC&D System (automated system designator D035) consists of nine subsystems identified with suffixes A, B, C, J, K, L, R, S, and T. D035A, the Item Manager Wholesale Requisition Process (IMWRP), is the heart of the requisition processing system; the other subsystems support related functions.

The SC&D System is comprised of nearly 2,000 programs (see Table 6). Most of the programs are written in COBOL and IBM CICS COBOL (53.6 percent) with the remainder written in IDEAL (37.9 percent) and other languages (8.4 percent). In total, there are more than two and one-half million lines of source code.

Some 307 input files are received from various systems; 113 of these are received on a daily basis. Nearly 400 output files are generated and sent to various systems; 230 of

Table 6
The Nine Subsystems Vary in Size and Programming Language

Subsystem designator	Number programs	Lines of Code (LOC)	Average LOC per program	COBOL	COBOL CICS	IDEAL	Others	Database
D035A	270	589,000	2,181	106	52	90	22	CA DataComm/DB
D035B	66	71,000	1,076	66	0	0	0	Flat Files
D035C	196	104,000	531	20	176	0	0	VSAM/Flat Files
D035J	212	343,000	1,618	102	4	106	0	CA DataComm/DB
D035K	487	725,000	1,488	258	5	190	34	CA DataComm/DB
D035L	64	110,.000	1,718	47	0	11	6	CA DataComm/DB
D035R	235	340,000	1,447	73	2	141	19	CA DataComm/DB
D035S	220	286,000	1,300	58	4	92	66	CA DataComm/DB
D035T	176	302,000	1,715	60	0	100	16	CA DataComm/DB
Totals	1926	2,760,110	1,433	790	243	730	163	

Note: Adapted from *Maintenance Analysis of the Stock Control and Distribution System* (p. 6), KPMG Peat Marwick Mangement Consultants, February, 1993.

these are generated on a daily basis. D035A, the component of primary interest in the investigation, receives 123 input interface files from 39 systems and generates 161 output interface files for 57 external systems.

A significant aspect of the SC&D System is the amount of time spent on maintenance. According to a report prepared by KPMG Peat Marwick in 1993, from August 1989 to November 1992, the mean number of monthly hours spent on maintenance was 11,550; the range was 4,000 to 26,800 hours. Dividing the mean value by 160 (a 160-hour person month) equals slightly more than 72, indicating the average number of full-time people engaged in maintenance. The report also indicated there was an additional enhancement and modification backlog of 206,511 hours, enough work for 108 full-time people for one year.

A Forward Engineering Model

It is clear from the literature that reverse engineering depends on, among other factors, knowledge, skill, and experience with forward systems engineering. It seemed appropriate, therefore, to begin work on a reverse engineering process model by first describing a general forward engineering model.

Most forward engineering process models begin with an activity alternatively called *requirements analysis*, *requirements acquisition*, or *requirements definition*. The activity is most closely related to the functional user. Most models end with an activity called

implementation. The implementation activity is most closely related to the hardware on which the completed system will operate.

A simplistic view of forward engineering identifies all other activities occurring between the requirements and implementation as design. This simplistic view is not sufficient for understanding reverse engineering because it omits too many important details. The design part must be expanded in order to understand the overall process.

Figures 10 through 14 are a low-level IDEF0 model of a generic forward engineering process. One process (A2) is decomposed to the next lower level in Figure 14.

Essential Points in the Forward Engineering Model

In the early phases of forward engineering, functional skills are the critical resource. As system development moves closer to design and implementation, technical skills begin to play a more important role. The "mechanism" flows (the flows entering the bottom of the processes in Figures 10 through 14) show the shift in the means used to perform forward engineering.

At a high level of abstraction, two distinct knowledge classes associated with software development are observed: domain knowledge and technical knowledge. Domain and technical knowledge are shown as “control" flows (the flows entering the top of the processes in Figures 10 through 14). The literature supports this observation.

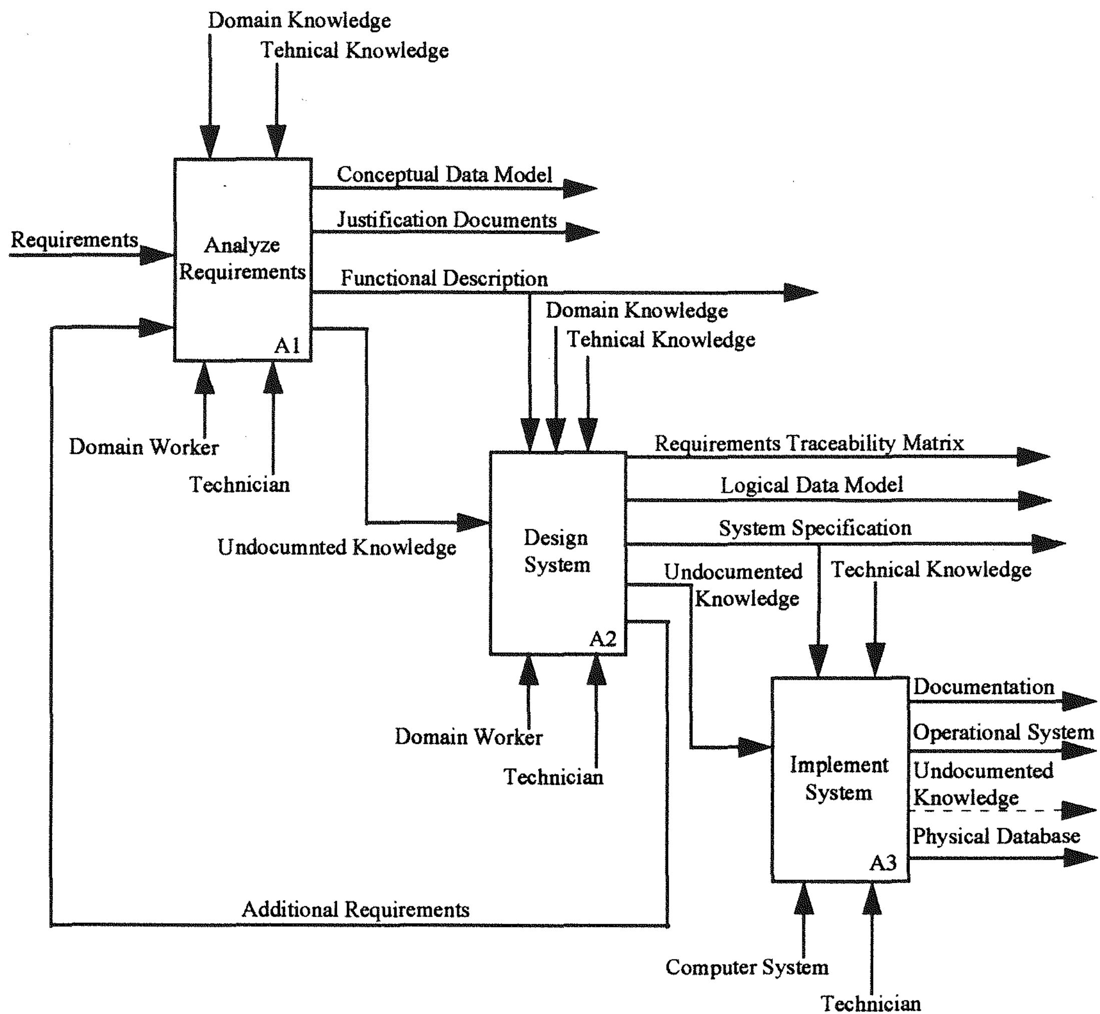


Figure 10. Forward engineering process model A0 diagram.

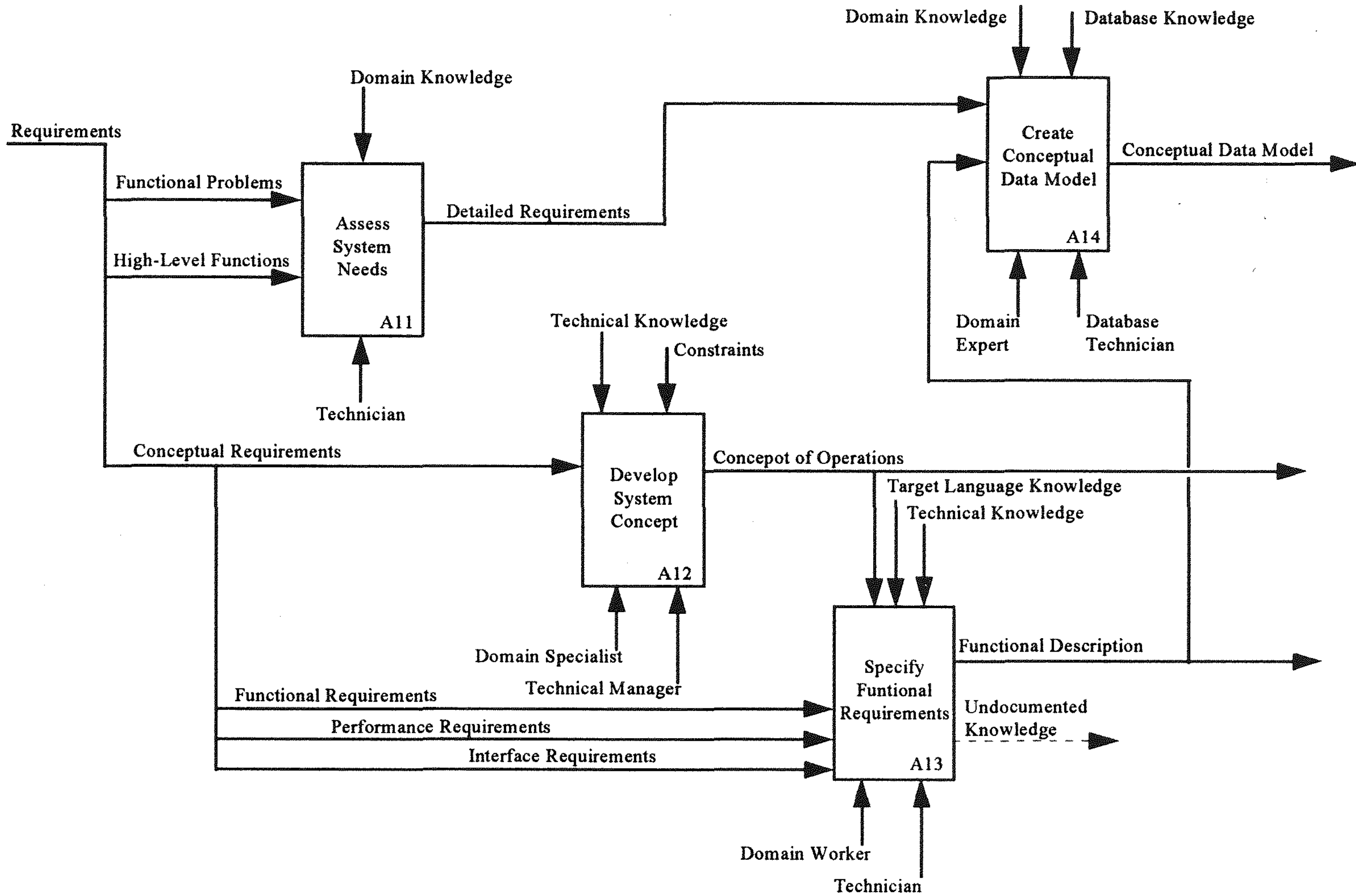


Figure 11. Process A1 decomposition.

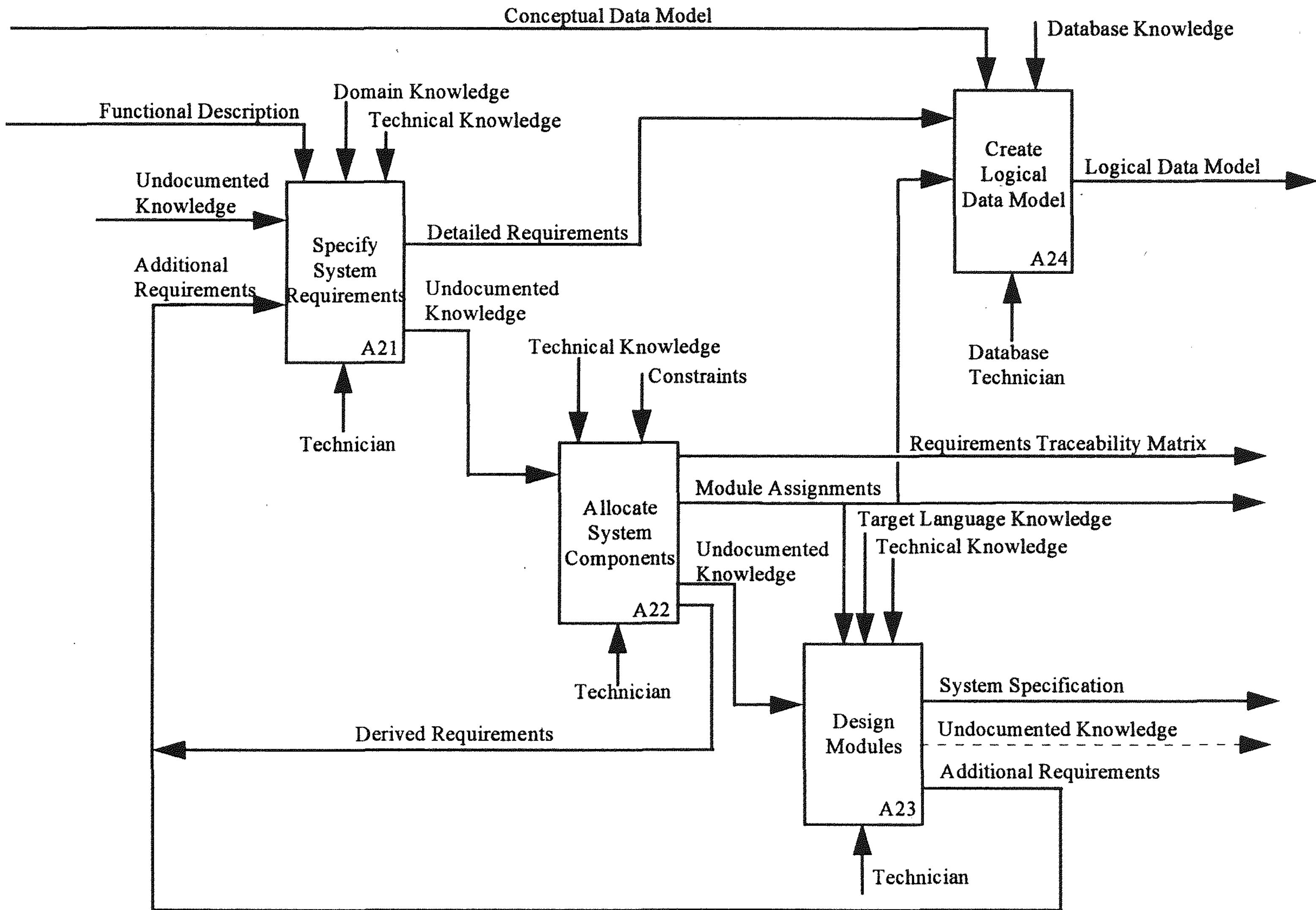


Figure 12. Process A2 decomposition.

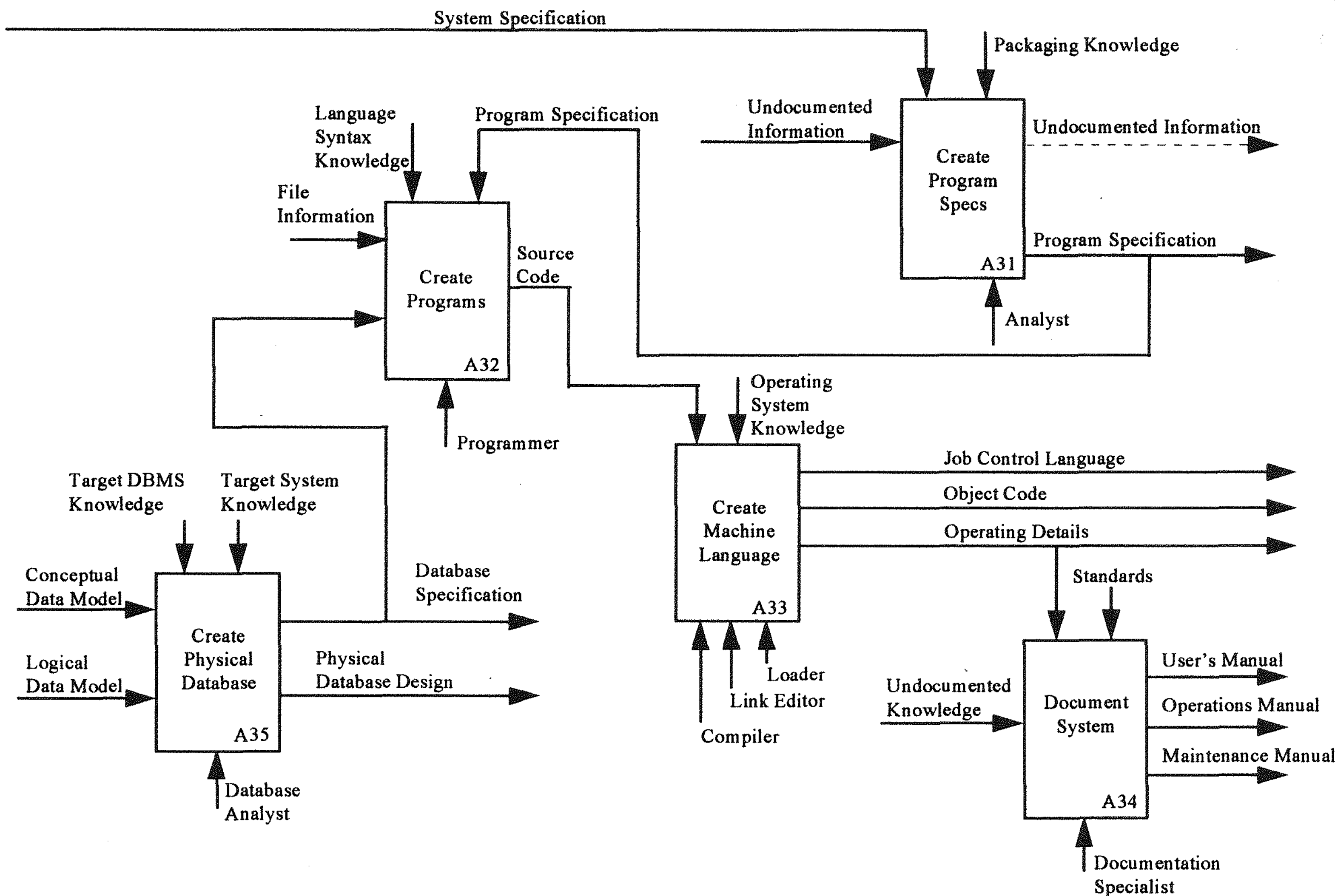


Figure 13. Process A3 decomposition.

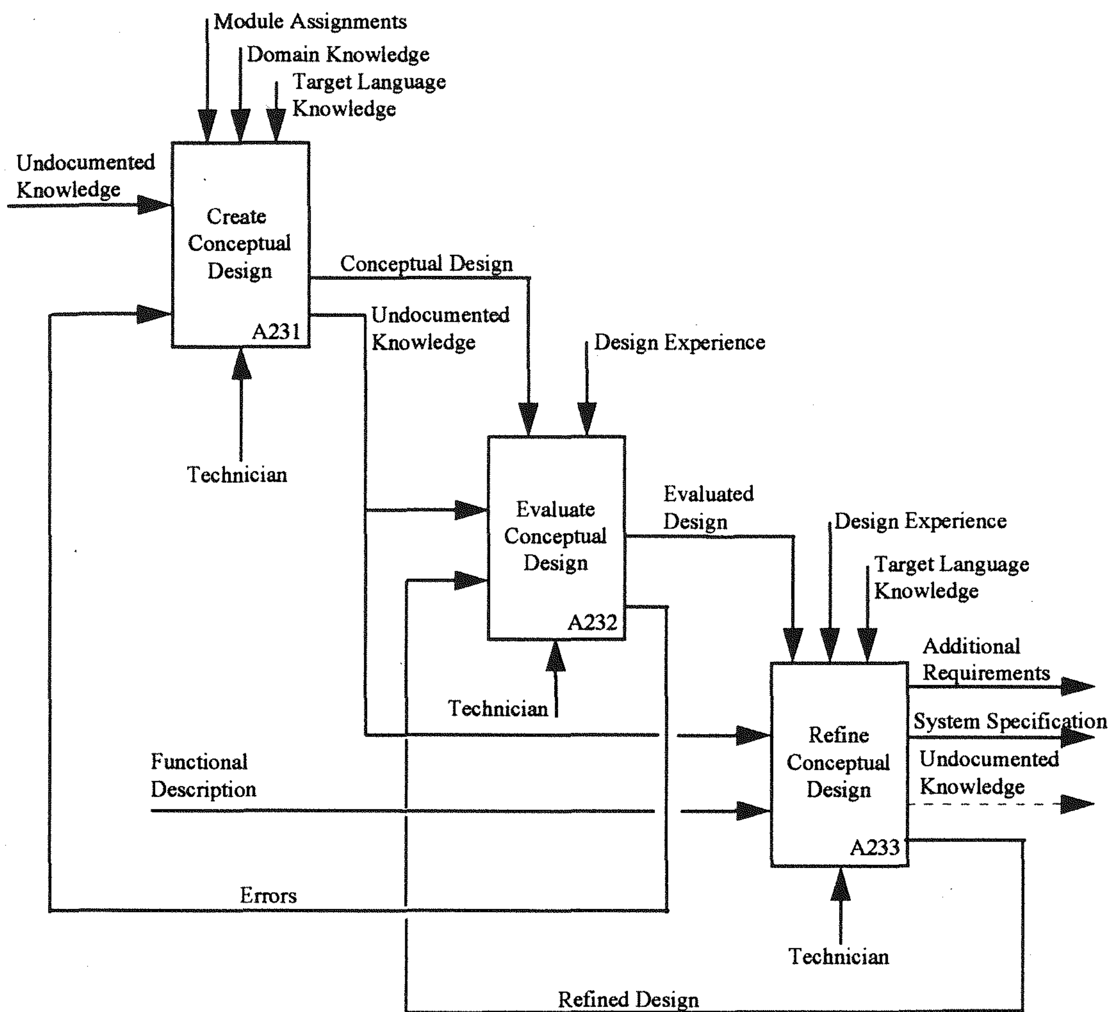


Figure 14. Process A23 decomposition

The major transition points in the forward engineering model occur between the Requirements and Design phases and between the Design and Implementation phases. These transition points are traditionally marked by the delivery of a documentation product representing the end of one phase and the beginning of the next. However, the forward engineering process is continuous, and the documents produced at the transition points are essentially snapshots of the status of the process at a point in time.

The forward engineering model diagrams include a flow identified as "undocumented knowledge." This flow represents knowledge used in the forward engineering process, but not included in the documentation.

An observation suggested by the forward engineering model is the transfer of knowledge between domain specialists and technicians during software system development (see Figure 15).

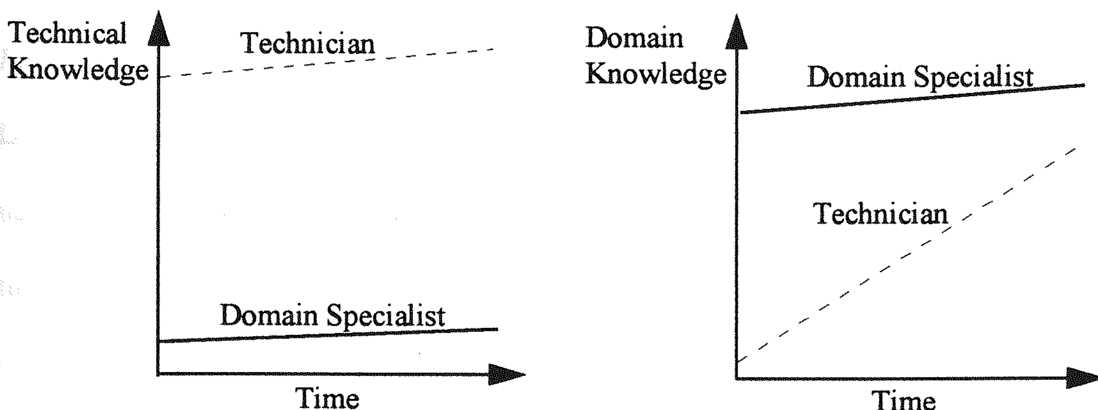


Figure 15. Software system development knowledge transfer.

The left side of Figure 15 represents the domain of technical knowledge and portrays relative levels for technicians and domain specialists. Knowledge levels remain relatively constant, but not flat. It is assumed that both groups slightly increase the overall level of technical knowledge as a result of software system development.

The right side of Figure 15 represents domain knowledge. Domain specialists have a high level of domain knowledge at the start of a software system development effort, and their knowledge increases slightly over time. This increase is a result of the thought and study given to the processes that are modeled for implementation in a management information system. The most significant change is shown for the rapidly increasing level of domain knowledge gained by technicians. In order to make the transition from a nearly pure domain model to a nearly pure technical model, technicians must acquire sufficient domain expertise to translate functional processes and domain objects into the imperfect world of implementation technology.

Information Loss in Forward Engineering

Documents produced during forward engineering do not contain the actual knowledge used to develop software. Documentation produced at the end of a phase includes only the results of analysis or design work; intermediate activities and decisions are not recorded.

There are three types of information loss during forward engineering (Brown (1983):

1. Closure - When application domain information is translated to another form (i.e., a specification), informal knowledge is lost because of the closed body of text.
2. Idealization - Simplifications of the application domain are made for reasons of conciseness and cogency of the specification. For example, business rule exceptions may be ignored to make the specification concise and uncluttered.
3. Domain - Program representations involve concepts from the application domain, and are not often represented in code. These concepts may only be known informally by the system user.

One of the problems with documentation is that it is difficult or impossible to record all the knowledge gained during systems analysis and design. Continuous informal communication usually occurs between domain experts (users) and technicians (developers). After technicians have achieved a basic level of domain understanding, they gradually expand their knowledge by forming specific questions and assimilating the answers provided by domain specialists. This interaction may constitute the bulk of informal communication.

Much of this informal communication, especially in the initial forward engineering stages, is verbal and specifically aimed at facilitating the transfer of domain knowledge to technicians. Information transfer takes place at a low level of detail over a considerable period of time. Because the information is unstructured, it is difficult to organize in a

form suitable for inclusion in documentation. It is hypothesized that most of this information is never captured in the end-of-phase documentation.

"After the fact" documentation (i.e., documentation created after the system was designed and developed) supports the hypothesis that documentation does not contain actual systems development knowledge. When a software system has been completed, the loss of domain knowledge acquired during development is complete. The only source of information available is then the source code. Source code listings stored in binders marked "system documentation" are not uncommon.

The information lost during forward engineering may be summarized as follows:

1. Non-procedural business knowledge used to make decisions.
2. Problem specification known informally by the analyst or programmer (undocumented knowledge).
3. Design justification (reveals how the implementation solves the problem contained in the specification).
4. Design decisions based on the problem of representing the application domain in systems constrained by the realities of imperfect technologies and imperfect programming languages.

A Model of the Reverse Engineering Process

Conceptually, reverse engineering is the opposite of forward engineering. Therefore, a possible starting point for a reverse engineering model can be created by reversing the forward engineering process model.

Figure 16 is an A0 diagram for a reverse engineering process model. This model was created by reversing the forward engineering process model inputs and outputs and deleting some of the mechanisms and controls. It should be clear even from this high-level diagram that simply reversing the forward engineering process is not an adequate approach for developing a reverse engineering methodology. If the decomposition were continued, the result would be processes that could not be implemented. It could be argued that the problem is the modeling technique rather than the process being modeled. The modeling technique, however, is arguably flawed because it allows the construction of a process model ultimately decomposing to a series of small, unsolvable problems.

Differences in Forward and Reverse Engineering

The forward engineering model previously described would elicit confidence from most information system technicians because it is based on a substantial base of successful experience. An information technician is reasonably confident that a software system will result from executing the activities of the forward engineering model.

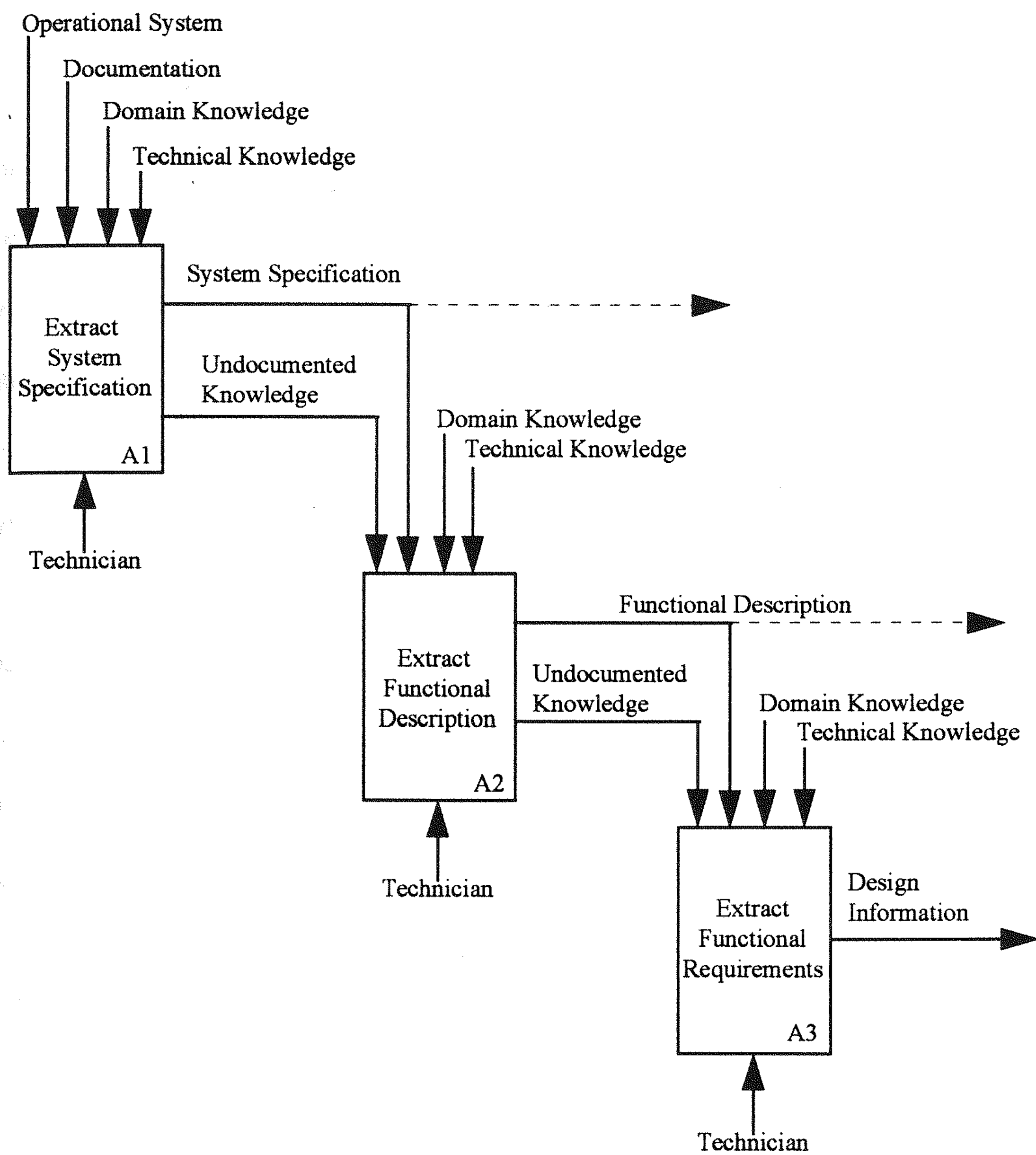


Figure 16. Reverse engineering A0 process model diagram.

This confidence is absent in the reverse engineering model. The difficulty of understanding source code written by a third person, even if it is well-structured and documented, is evident to any maintenance programmer.

The objective of forward engineering is to implement a computer information system. The steps between requirements and implementation are coincidental and are not essential to writing programs (although they simplify the process). Systems *can* be developed by the "just start coding" approach. There is no equivalent "just start uncoding" approach for reverse engineering. The complexity of reverse engineering, even for relatively simple systems, can be overwhelming.

During forward engineering, a finite set of functional requirements is translated or transformed from a well-specified problem domain to a well-specified technical domain. During the transformation, the domain aspects are gradually "lost" and the design becomes purely technical. During forward engineering there is a clear target, implementing a system, and the possible implementations are only limited by technology.

During reverse engineering from the technology-based model to the domain-based model, there are a finite but large number of possibilities. The target of reverse engineering is not clear except for a broad range of possibilities within a specific application domain.

In the forward engineering process, the path taken through the technical possibilities is immaterial as long as the system created satisfies its intended purpose. In reverse engineering there is significantly less flexibility. Not only must a reverse path through the various design decisions be identified, it must be the same path. In other words, the reverse engineering process must identify the same design information or requirements upon which the implemented system was based.

Problems Associated with Reverse Engineering

Human capabilities and capacities are one of the most important problem areas associated with reverse engineering. Given that software development is a human activity, albeit supported by computer-based tools, it is obvious that reverse engineering is also a human activity.

A fundamental problem of computer-based reverse engineering tools is that computers are limited in how they can represent software internal complexity. Humans are able to understand software complexity without representing its internal structure. Humans are not efficient, however, at handling the large volumes of information associated with reverse engineering a system.

In the early years of data processing, programmers were in great demand because of their unique skills. As a result, many legacy system programs were coded by inexperienced programmers not adept at writing COBOL.

A reverse engineer must have extensive experience with top-down modeling techniques. It is easy to develop a high-level model that is so shallow it has no real content; it is also easy to develop a low-level model that contains so much detail it is incomprehensible. Knowing when the correct level has been achieved is the secret of good modeling; and extensive functional modeling experience is the secret of being a good modeler.

There are few people with in-depth knowledge of how legacy systems were constructed or what they do.

Reverse engineering ability depends, in part, on program and application domain familiarity and programming style.

A major part of reverse engineering is the problem of discovering human-oriented concepts of computational intent and assigning them to their realizations within a specific program or its context (see Biggerstaff, et al., 1994).

Human-oriented terms used to represent knowledge are succinct, ambiguous, informal, and intelligible to other humans. Computational-oriented terms are based on a narrow and restricted grammar and vocabulary. There is little or no connection between human-oriented terms and formal computer language.

As pointed out by DeBaud, et al. (1994), most reverse engineering begins by analyzing program structure with lexical, syntactic, and semantic rules. Debaud, et al. compared this with trying to read English knowing only the rules of English grammar.

As Pfrenzinger (1992) noted, the higher the target level on the reverse engineering scale, the less automatic and more manual the reverse engineering process becomes. Formal methods (e.g., mathematical models, wide-spectrum languages, and artificial intelligence techniques) are difficult for humans to understand, but they are suitable for automated reverse engineering tools.

Concepts from source code must be correlated across multiple perspectives. Reverse engineering is based on the ability to recognize, comprehend, and manipulate design decisions in source code. Causal connections between requirements and computer programs must be identified.

A reverse engineer must be able to explain each program and relate its structure and behavior, as well as its relationship to the application domain. The terms used to explain programs are significantly different from the language elements used to write the source code. A reverse engineer must be able to assign more meaning to program text than what is included the source code.

There are three levels of aggregation in reverse engineering: (a) low-level understanding of source code, (b) mid-level understanding of algorithms and data, and (c) high-level understanding of overall program function.

Syntactic knowledge of COBOL must be applied to uncover the semantic meaning of the source code. Sources external to the source code may be required to determine semantic meaning.

According to Munro (1992), a software system consists of the following elements: source code, JCL, databases, object code, documentation, design information, requirements details, specification details, knowledge of analysts and programmers who developed the system, and the knowledge and expertise of maintenance programmers. Many of these elements may be missing in a legacy system.

There may be operational functions in legacy systems that are not used because the results are incorrect, unreliable, or incomplete. To correct the problems, new code may added. As old code is seldom removed; the source code continues to increase in size and complexity.

Most legacy systems were not well engineered during development, complicating reverse engineering. As legacy systems were developed over many years, the code base is extremely large.

According to Sage (1993), it is difficult to describe large complex systems by structure, function, or purpose because these views are not mutually exclusive nor collectively exhaustive.

As noted by Ornburn and Rugaber (1992), program text is inherently ambiguous; it is difficult to identify the purpose of program structure without contextual information not found in text. A reverse engineer must be able to draw on a broader knowledge base, reconstruct the missing context, and determine the functional intent of the software design.

A reverse engineering methodology must provide a method for abstracting design information above the source code level as rapidly as possible. The volume of source code and the fundamental manual nature of reverse engineering dictate this principle.

COBOL statements have three features: semantic (what the statement does), syntactic (how the statement is formed), and lexical (rules by which elements of the syntax are formed). A reverse engineer must have a comprehensive understanding of these features.

COBOL, as a formal language, has syntactic content but not semantic content. However, there may be some semantic content depending on the names used for variables, e.g., ADD A TO B GIVING C and ADD BASIC-PAY TO OVERTIME-PAY GIVING TOTAL-PAY. Both

statements are syntactically correct, but the second has semantic content not present in the first. There is no 1:1 mapping between the syntax and the semantic content.

Textual material other than program source code can play an important part in reverse engineering if it is available, accurate, and current. System documentation is almost universally poor and must be used with caution during reverse engineering.

There are two kinds of system documentation: low-level physical implementation details and high-level conceptual overviews. Documentation between these two levels is rare. The probability of extracting an accurate model from existing system documentation is small. Small systems, which are relatively easy to model, tend to have the best documentation. Large systems, which are more difficult to model, tend to have inadequate documentation. The quality of system documentation appears to be inversely proportional to its value as a reverse engineering tool.

Because of the number of input and output files coming from and going to other systems, interfaces play a crucial role in reverse engineering in the Air Force logistics systems domain. Interfaces tend to have the same problems associated with source code and documentation. Interfaces are nominally described in Memoranda of Agreements (or Interface Control Documents), but the agreements tend to be short on information content, outdated, inaccurate, and in conflict with the actual content of the interface files.

The following considerations are relevant to interfaces:

1. Input Source - The input source may not be easily identifiable because of an intermediate communications system that collects transactions from multiple sources.
2. Sending System - It may be important to know what process in the sending system produces the interface file. For example, many interface files are copies of intermediate files produced at a particular point in a batch process as a matter of convenience.
3. Receiving System - In the reverse engineering environment, two receiving systems are considered: the target system being reverse engineered and the system that receives the output interface files from the target system. In both cases it is important to know how a receiving system uses the data in an interface file, i.e., Is all the data used? Is some of it not needed?

A Formal Method of Reverse Engineering - Clean-Specify-Simplify

Lano, et al. (1993) described object-oriented methods and tools to reverse engineer COBOL application programs to program specifications. The basic concept (identified here as Clean-Specify-Simplify) is to recover design and function from programs by creating object-based abstractions. The main process is transformation: COBOL source code is transformed to Uniform; Uniform is transformed to functional description language; and functional description language is transformed to Z specification language. Byproducts of the process include data flow diagrams, entity-relationship diagrams, and call-graphs.

The method is largely automated and is comprised of three stages:

1. Stage 1 - Clean. Restrict original language to a small subset of permissible constructs. Translate the source code to an intermediate language (Uniform), eliminating redundant language constructs. Translate asserted relationships between data values into statements about invariants in the program's run-time behavior.
2. Stage 2 - Specify. Create prototype objects by grouping associated variables using data flow diagrams for guidance. Object-based entity descriptions consist of attribute lists and initial values. Associated operators are not yet included. Split code into phases. Phases are "maximal logically connected sections of code within which no files are opened or closed, or have their read/write status changed" (Lano, et al., p. 15). Phase functionality is automatically obtained and transformed to intermediate functional language.
3. Stage 3 - Simplify. Incorporate abstracted functional descriptions into the outline objects as descriptions of their operations. Print a full specification in Z or Z++ using the object-based abstraction and associated textual documentation.

The basis for the Clean-Specify-Simplify approach to extracting functional information from source code as described by Breuer and Lano (1991) is based on the belief that some concept of functionality can be derived from looking at program input and output relative to the internal data structures specified by the program. Detailed functionality, however, can only be discovered by line-by-line source code analysis

Positive aspects of this technique are:

1. It is designed specifically for COBOL.
2. It recognizes the need for a formal reverse engineering plan.
3. It recognizes the need for interaction with maintainers rather than relying solely on source code.
4. It establishes higher-level abstractions in Stage 1 by finding files described in the file section and environment division and representing them as objects. Flags, counters, and other information related to each object are added as they are identified.
5. If the recovered objects are viewed as opaque, abstraction is achieved by eliminating implementation details. Real (higher-level) abstractions of object semantics can be produced by replacing captured functionality with more general specifications.
6. It is related to the process view of systems because the global functions identified as object class methods correspond exactly to the processes.

Negative aspects of the technique are:

1. It is meant to produce only program-level information. The objective is to produce program specifications from source code and to preserve the specifications in sufficient detail to recreate the original program.
2. It is designed to support maintenance, reengineering, and reuse rather than design recovery.

3. COBOL code is abstracted to produce explicit mathematical descriptions of functionality and object classes representing the application design. These representations are difficult to understand.

A Structural Approach to Reverse Engineering - Program Schematics

Lerner (1991) developed a reverse engineering approach based on program schematics. Lerner described the technique as a reengineering approach to decompose a single program with several applications into several programs, each for a single application. The source program used in the example--a six-year old, 1,800-line BASIC program with over 300 transfers of control written for a Commodore 128 computer--was reengineered to an IBM personal computer with MS-DOS BASIC with a 64,000 bytes memory. The source code listing was not included with the description.

Lerner (1991) detailed a six-step reverse engineering enactment process. The process evolves through four steps of creating documentation "in-the-small" and through two steps of creating documentation "in-the-large." Creating documentation in-the-small is the process of dissecting a program into formal units, declaring names of these units, creating functional units, and defining the immediate impact environment of the functional units. Creating documentation in-the-large involves declaring and semantically describing linear program circuits and system applications.

Step 1 - Dissect Program into Formal Units. Six rules describe the processes performed in this step:

1. A formal unit is a segment of code which starts with a program statement to which control is transferred from anywhere in the program. Statements such as GO TO, GOSUB, CALL, and PERFORM transfer control; the objects of these verbs are formal units.
2. A starting statement label (or a line number) defines a unit entrance. The entrance is identified by a "0."
3. A formal unit ends where another formal unit starts.
4. A subroutine formal unit ends with a return-like statement.
5. A non-subroutine unit does not have a return-like statement.
6. A unit program statement that transfers control to another formal unit is called a unit-exit. Exits are numbered in sequence.

Step 2 - Declare Names of Formal Units. Formal units are numbered for identification. Non-subroutine units are identified by one set of numbers (e.g., 100-499). Subroutine units are identified by a different sequence of numbers (e.g., 500 and up).

Step 3 - Create Functional Units (see Table 7). There are three options for creating functional units: (a) a formal unit and a functional unit have the same segment of code, (b) a formal unit is dissected to create several functional units, and (c) several consecutive formal units are combined to create a single functional unit. The name of a functional unit

is identified from the code. The name may also be derived from program remarks, if they are present, or from a maintenance programmer.

Table 7
List of Functional Units

Non-subroutine units		Subroutine units	
Unit number	**Unit name**	**Unit number**	**Unit name**
100	Heading	500	Record printout
101	Dimensions	501	Change file name
102	Disk	502	Part/board list
103	Read disk	503	Array is full

Note. Adapted from "A Standard Approach to the Process of Re-engineering Long-lived Systems," by M. Lerner, 1991 (Summer), *CASE Trends, 3,* p. 19.

Step 4 - Define Immediate Impact Environment of Functional Units. Units that transfer control immediately to a particular functional unit create an input environment. The input environment describes the impact of many on one. Those units to which a particular unit transfers control create an output environment. The output environment describes the impact of one on many. A combination of the immediate impact environment of each unit with the segment of code that belongs to the unit creates schematic documentation in-the-small, and is the basis for local analysis. In-the-large documentation is used for global program analysis.

Step 5 - Declare Linear Program Circuits. A linear circuit (LC) is a succession of at least three non-subroutine units. An LC starts with the first unit and ends with the last unit. Declaring LC starts with mapping a network of non-subroutine functional units (see

Figure 17 for a portion of this map). Figure 17 contains three types of non-subroutine units: (a) a transiting unit - one-to-one unit (e.g., unit 101); (b) a branching unit - one-to-many unit (e.g., unit 108); and (c) a rooting unit - many-to-one unit (e.g., unit 107). The first unit can be the beginning unit of the program, a branching unit, or a transiting unit. The last unit may be the end unit of the program, a rooting unit, or a transiting unit. The first LC must create a trunk of the tree of LC. The trunk starts from the beginning of the program and ends at the program end.

Figure 18 is the program LC 001. It starts with beginning unit 100 and ends with end unit 128. This LC represents the trunk of the tree of LC. After the trunk was defined, the next LC starts from a branching unit, which belongs to the trunk and at the same time is the closest to the program end. Branching unit 108 is the closest to the end unit from which the next LC start.

Proceeding backward from the end of the program, all other LC are declared until each functional unit belongs to at least one LC. Each LC represents a certain mode of system operation. The purpose of this mode is described by the LC name. Some of the 29 declared LC from the sample program are listed in Table 8. Trees of subroutine units, if complicated, can be described by LC to make them readable.

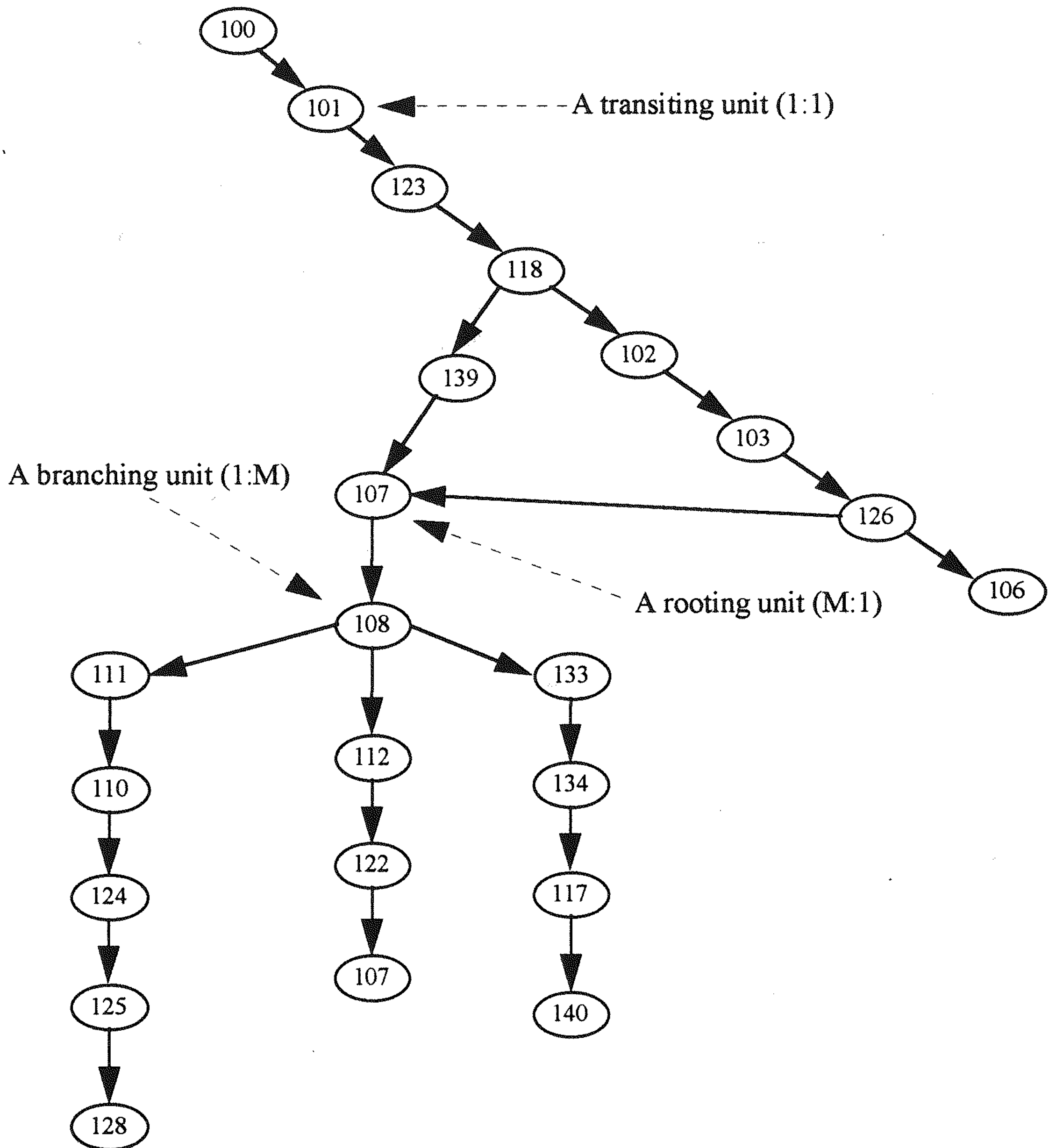


Figure 17. Network of non-subroutine units.

Note. Adapted from "A Standard Approach to the Process of Re-engineering Long-lived Systems," by M. Lerner, 1991 (Summer), *CASE Trends, 3,* p. 20.

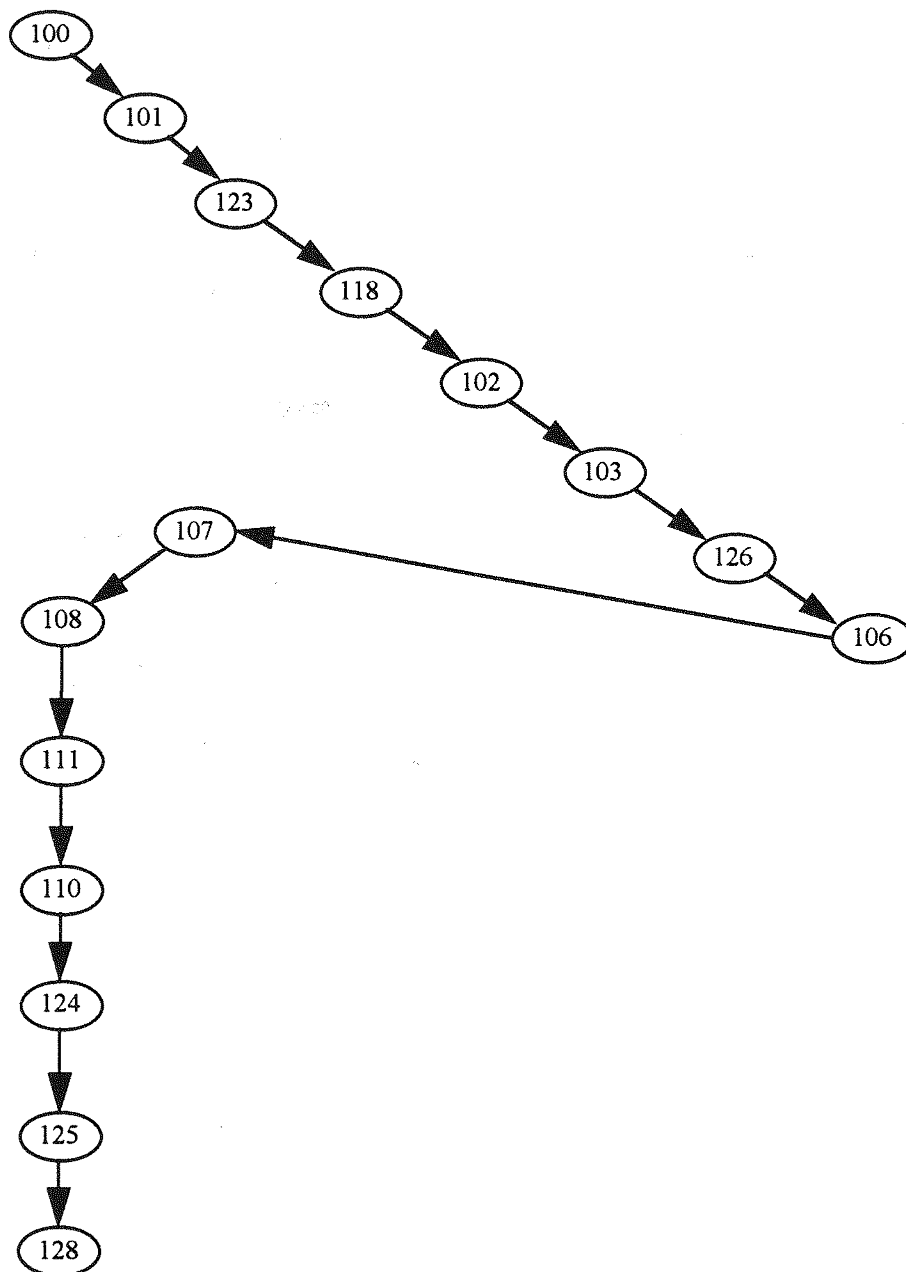


Figure 18. Trunk of the tree LC.

Note. Adapted from "A Standard Approach to the Process of Re-engineering Long-lived Systems," by M. Lerner, 1991 (Summer), *CASE Trends, 3,* p. 20.

Table 8
Linear Circuits

Number	Name	First unit	Last unit
001	Create genetic matrix	100	128
002	Create LC name	110	155
003	LC name amend	156	161
004	LC name delete	154	158

Note. Adapted from "A Standard Approach to the Process of Re-engineering Long-lived Systems," by M. Lerner, 1991 (Summer), *CASE Trends, 3,* p. 20.

Step 6 - Define System Applications. A program application is a family of LC that performs a specific data processing task defined by the user. Defining which LC belong to a program application starts with mapping all branching and rooting non-subroutine units, which are the first and the last units defining LC. This map is used to declare program applications, each being a family of interrelated LCs.

In the sample, Application 1 consists of six LC--001, 007, 008, 010, 011, and 012 (see Table 8). LC 001 consists of 23 non-subroutine units--100, 101, 102, etc. (see Table 9)--and 40 subroutine units (not shown in the example). Extracting these functional units from the original program resulted in a collection of all program statements involved in application 1. This procedure was repeated for the other five LC.

Table 9
Application 1 - LC Components

Linear circuit number	Linear circuit name
001	Create genetic matrix
007	Print genetic matrix
008	More create?
010	Delete link
011	Amend link
012	Start genetic matrix

Note. Adapted from "A Standard Approach to the Process of Re-engineering Long-lived Systems," by M. Lerner, 1991 (Summer), *CASE Trends, 3,* p. 21.

Positive aspects of the approach are:

1. The technique is based on physical inspection of the program source code.
2. Schematics are a logical, straightforward technique for extracting functional applications from the source code.
3. Schematics are primarily graphics-based and reasonably easy to prepare and use.
4. Schematic mapping offers a method for abstracting to a higher level than program code.

Negative aspects of the approach are:

1. The schematic technique was apparently designed for a dialect of BASIC with limited capabilities.

2. COBOL programs to be reverse engineered are larger and more complex than BASIC programs performing the same function; schematic diagrams for these programs may be too large to work with easily.
3. COBOL programs use labels, rather than line numbers, as addresses for GO TO, GOSUB, and PERFORM statements.
4. COBOL syntax is more complicated than BASIC--variable and file names are longer, and data and procedure divisions are separate program components.

A Program Understanding Approach to Reverse Engineering - DESIRE

The DESign Information Recovery Environment (DESIRE) is a program understanding assistant system developed by Biggerstaff, et al. (1994). According to the developers, two key properties distinguish this design recovery model from similar models:

1. Use of Informal Information - The model exploits multiple kinds of information. It uses informal information from outside of the sphere of programming languages and exploits a human-oriented, associative style of retrieval and analysis.
2. Use of a Domain Model - The model exploits multiple sources of information. It uses a domain model to help the software engineer understand and interpret foreign systems. The domain model is a knowledge base of expectations (i.e., patterns of program structures, problem domain structures, language structures, naming conventions) that provide frameworks for code interpretation. These frameworks can be used to replace missing design information.

DESIRE uses the Common Lisp Object System (CLOS). Key structures in the source code are represented in CLOS by object classes. Domain model classes are used to search for instances in the code and to bind instance variables to the domain key structures, subject to analytic approval. The instance variables point to the segments of code that implement the domain object. The first step is to create a set of instances of the idiomatic structures expected. Each instance can be bound to source code in one of two ways--direct or indirect.

In direct binding, a pattern instance is bound directly to a segment of code. Implementation is via a linguistic idiom representing the expected linguistic form of a conceptual abstraction. The idiom is implemented as a set of regular expression patterns that match the various natural language forms in source identifiers or comments. When an unrecognized expression of the conceptual abstraction is encountered, it is added to the domain model. In indirect binding, a pattern instance is bound indirectly through a sub-instance (through a close match of the substructure to the program code).

A linguistic idiom expresses natural language tokens generalized into search patterns associated with key data structures; data object idioms express the substructure relationship within complex data structures.

Idiom pattern matching to code is inexact. An automated aide is typically able to produce only partial matches. Some instances of the idiom are unbounded and some elements of

the structure are unexplained; this part of the interpretation work must be completed by a software engineer.

The conceptual abstraction instances produced by design recovery go beyond what can be represented in programming languages (Biggerstaff, 1989). They are represented in both rigid formal terms and informal and flexible terms. Biggerstaff said these artifacts are not simply optional, informal additions to the formalisms expressed in the programming languages, but complementary representations necessary and critical to the mental structuring and assimilation of the final design.

A considerable amount of design information cannot be formally captured in program source code because programming languages do not contain the necessary constructs to express such information as the informal conceptual abstractions behind the code (Biggerstaff, 1989).

Biggerstaff, et al. (1994) identified two general tasks required when attempting to assign concepts to code:

1. Identify which entities and relations out of the often overwhelming numbers in a large program are important.
2. Assign important entities and relations to known (or newly discovered) domain concepts and relations.

Task 1 relies heavily on generic, formal information such as data structures, functions, and calling relations, as well as informal information such as grouping and association clues. Task 2 relies more heavily on domain knowledge (i.e., knowledge of the problem domain entities and typical application architectures and relationships).

Task 1 uses generic knowledge to infer that statements are related to one another in a non-casual way because they are: (a) grouped together (proximity), (b) bracketed with blank lines, (c) exhibit a strong surface similarity among many of the formal and informal tokens, and (d) exhibit coupling via common tokens among several definitions.

Task 2 features suggesting concept assignments are: (a) natural language token meanings, (b) occurrence of closely associated concepts, (c) individual relations paralleling those in the model, and (d) the overall pattern of relationships in the model.

Positive aspects of the approach are:.

1. The importance ascribed to informal external information in augmenting reverse engineering techniques.
2. The concept of a domain model that captures information to support the reverse engineering process.
3. Non-exclusive reliance on the automated system to recognize and identify program plans.

Negative aspects of the technique are:

1. The technique is based on the program plans approach.
2. The technique is primarily an automated assistant tool.
3. The domain model is established as a knowledge base of expectations and is limited by difficulties associated with collecting knowledge from domain experts.
4. Domain knowledge must be stored in a format that can be processed by a computer.
5. The benefit of available domain information is limited by the form in which it is stored.

A Data-oriented Reverse Engineering Technique - Component Extraction

According to Lanubile and Visaggio (1993), business systems are data-oriented because most of their tasks are related to manipulating large amounts of data stored in a database. They maintained that most of the knowledge needed to understand business systems, both conceptual domain and implementation software models, is contained within the system data.

Lanubile and Visaggio (1993) proposed a method to identify and extract environment-dependent components and domain-dependent components from a business application system. Environment-dependent components depend on the technological environment of a system and usually consist of basic operations on a database, report production, display of interface maps, or user machine dialogue. Domain-dependent components characterize a class of problems in the same application domain and typically consist of computational

formula or business rules. Differentiating between these components is advantageous for adaptive maintenance or platform migration.

The component extraction technique is described as part of a reverse engineering process model for data-oriented applications. Lanubile and Visaggio (1993) claimed the detailed knowledge of external inputs and outputs increases the application domain knowledge and provides clues for understanding the procedural code.

The component extraction technique is comprised of two phases: data recovery and function recovery. The data recovery phase is based on a reference information model applied to all information systems in the same class of problems. The reference information model is expressed in terms of entities, hierarchies of entities, and meaningful relationships between entities by using entity-relationship diagrams. Data recovery provides knowledge about external data. By analyzing file declarations, reports, and input/output maps, the following distinctions are made:

1. Conceptual Data - Data associated to an entity or relationship of the reference model.
2. Control Data - Flags used to control program logic.
3. Structure Data - Fields used to build data structures independent of the programming environment.

The reference model provides a template for classifying conceptual data in the code declaration. Data derived from other primitive conceptual data is derived data. The

computation formula or the business rule for derived data is recorded in a data dictionary. Identifying derived data is important because these values represent expectations about the existence of transform functions in the source code. Modeling data in terms of entities and relationships leads to expectations about source code components containing basic data structure operations: create, read, update, and delete.

In the function recovery phase, programs are separated into distinct components performing a single function (see Figure 19).

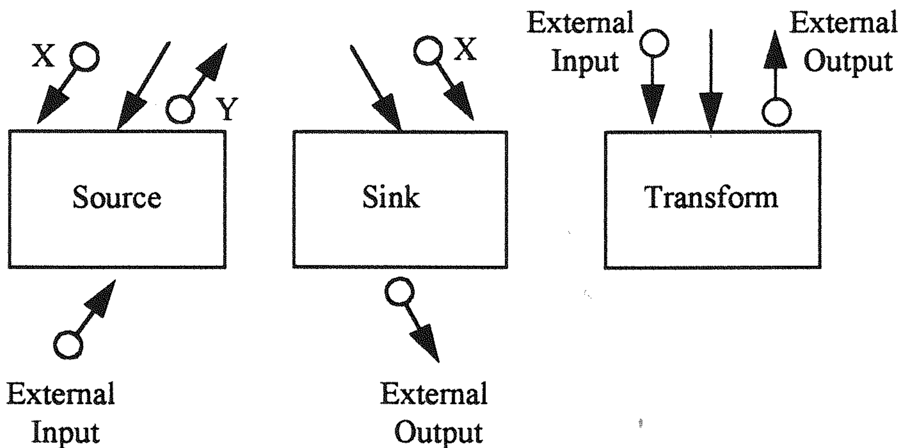


Figure 19. Three kinds of recovered components.

Note. Adapted from "Function Recovery Based on Program Slicing," by F. Lanubile and G. Visaggio, 1993, *Proceedings of the IEEE Conference on Software Maintenance,* p. 397.

The components and their functions are:

1. Source Module - Obtains information from external sources (i.e., `READ-NEXT-RECORD`, `OBTAIN-TRANSACTION`).

2. Sink Module - Sends data to external device (i.e., `ADD-RECORD`, `PRINT-REPORT`).
3. Transform Module - Transforms input data into some other form (i.e., `COMPUTE-INSTALLMENT-AMOUNT`, `VERIFY-LOAN-REQUEST`).

Source and sink modules are environment-dependent components; transform modules are domain-dependent components. Transform modules also correspond to processes in a data flow diagram.

Extraction is the process of capturing all statements that directly prepare data and execute the input or output source.

Lanubile and Visaggio (1993) implemented their component extraction approach by program slicing. Program slicing finds portions of source code that directly or indirectly affect the values of variables at a given instruction. The slicing methodology was modified to include only statements that characterize source, sink, and transform modules. A commercial tool, VIA/Renaissance, was used to perform the slicing.

Lanubile and Visaggio (1993) reported the recovered slices from large programs can be unmanageable. Their function recovery process was not completely automated because the slicing algorithm was not fully supported. The tool had drawbacks when program

slicing was applied to files with multiple record types or when the slicing criterion did not contain the variable of the slicing criterion itself.

Positive aspects of the technique are:

1. Based on the fundamental data processing model--input is processed and converted to output.
2. Stresses the importance of understanding external input and output as a way of understanding internal processes.
3. Can be manually implemented.
4. Theoretically separates input and output slices related to the environment from transform slices related to the application domain.
5. The number of slices to be analyzed for reverse engineering is reduced by the number of input and output slices.

Negative aspects of the technique are:

1. Centers on recovering executable code components suitable for reuse in another application in the same domain.
2. The recovered code slices are at the program level; there is no abstraction to a higher level.
3. The number of slices recovered from a large program can be difficult to manage.
4. Slices may be too complex to easily understand.

A Data Repository Approach to Reverse Engineering - System Description Database

Ostrolenk, Tobin, Altes, and Younger (1993) described a reverse engineering repository called the System Description Database (SDDB). The SDDB was developed as a part of the European REDO project (van Zuylen, 1993).

SDDB was required to store and interrelate information about an application, including program source code, life cycle documents, diagrams, notes, and links created by a reverse engineer, application data models, and formal specifications. Source code included compilable program modules, job control language (JCL) scripts, database management system schema, and transaction specifications (i.e., IBM Customer Information Control System (CICS) tables).

Six categories of source code information were identified for SDDB:

1. Original Source Text - The layout and appearance of source code.
2. Abstract Syntax - Source language constructs used to express program functionality.
3. Statement Semantics - Defined in terms of requirements for a compiler in a particular language and modeled as an abstraction from textual and syntactic constructs available in a particular language.
4. Data Usage - Includes the type and location of explicit references to variables, constants, and files.
5. Control Flow - Information affecting the sequence of statements, procedures, and program and job execution.
6. Data Definitions - Types of all variables, constants, and files.

The SDDB subset for COBOL representation defined approximately 120 entities and 160 relationships. An additional 50 relationships were added to represent calls to the database management system. A significant amount of database complexity results from the need to store syntactic representation of the source code.

A data dictionary stores information about records, arrays, COBOL level-88 entries, file devices, paragraphs, sections, alphabets, and operating system constants and switches. These components are stored as subtypes of the entity SYMBOL. Each SYMBOL is associated with a type definition and is related directly to the program in which it is described.

COBOL paragraphs are modeled as entities composed of statement sequences. Other statement sequences are stored in the same manner (e.g., THEN and ELSE clauses of IF statements, ON SIZE ERROR exception clauses).

Operating system environment software (e.g., JCL, file handlers, and transaction processing monitors) is modeled in the conceptual schema. Information is stored in the form of explanations of *what* the software does when a request is issued by an application. System software service calls are usually formatted by CALL statements to a specific module with a parameter list. Database interactions are modeled in a similar fashion.

Documentation such as specifications, design documents, and user documentation are stored in hierarchical structures as sections, subsections and paragraphs. Document version control is also provided.

Notes and links provide powerful mechanisms to support documentation. Links are a means of connecting two entities stored in the database; notes provide a means to annotate these entities.

Notes provide a means to incrementally add information to the database. Notes consist of unstructured text linked to entities in the database and are used to record information about the entity to which they are linked. When they are created, notes and links are "stamped" with the author's identity and creation date.

Other components of the SDDB are: database editor, query tool, source code browser, documentation browser, note tool, and link tool.

The note tool is considered a particularly significant piece of the SDDB. The process of reverse engineering is seen by Ostrolenk, et al. (1993) as one of incremental acquisition of knowledge in an iterative, rather than a linear, fashion. There is a need for a tool to be used to record knowledge acquired during this process. Notes can be used by individual maintainers as an aid to help them in their study of an application, and to record the understanding gained.

Positive aspects of the technique are:

1. The concept of a centralized repository for collecting information about an application.
2. The ability to collect information about various aspects of an application, including data, files, database management systems, and operating system calls.
3. The ability to store documentation related to an application on-line and to access it hierarchically.
4. The ability to store information about the types and locations of explicit references to variables, constants, and files.
5. The data dictionary capability for records and files (i.e., structures above the attribute level).
6. The links feature allows relationships to be established between any two objects stored in the database.
7. The notes feature allows unstructured textual knowledge to be immediately stored in the database.

Negative aspects of the technique are:

1. Oriented towards source code; the original layout and format of source code is retained in the database.
2. Retains details of program structure (e.g., the sequence of statements and procedures).
3. The data structure supporting the tool is large (120 entities, 210 relationships).

4. The tool has more reengineering features than reverse engineering procedures.
5. There is insufficient emphasis on abstraction to a level higher than source code.

Reverse Engineering Approach for Air Force Logistics Systems

Of the eight general types of reverse engineering techniques and methodologies previously discussed in Chapter II, none were considered to be ideally suited to the domain of Air Force logistics systems.

Software Physical Structure - This group of techniques focuses on code-level information; they do not provide an approach to recovering high-level functional information.

Knowledge-based Program Understanding - Without exception, these tools are still in the research stage. None has been successfully applied to a complex system.

Transformation - This group of techniques focuses on converting one form of language to another form with little or no human intervention. There is no change in the abstraction level of the source code.

Program Plans - These techniques are still in the experimental stage and operate only on simple programs.

Data Flow Diagrams - When recovered from program source code, these techniques stress physical program structure rather than functional data flows.

Functional Abstraction - These techniques are still in the research phase; none have been applied to complex systems.

Commercial Products - With the exception of data reverse engineering tools, tools on the market today are not true reverse engineering products. They are more properly identified as reengineering and maintenance tools.

Other Techniques - Two techniques are of interest from this category: manual code examination and the program information database. Other techniques in this group are either too complex to be applied to real world problems or are more appropriate for maintenance and reuse purposes.

Based on the literature review, evaluation of current research relative to reverse engineering, and detailed evaluation of five promising techniques, it was concluded that the most effective approach to reverse engineering for the Air Force logistics system applications is a form of manual code examination supported by a reverse engineering data repository.

The manual code review approach reflects the inherent ability of a human to abstract complex software directly into higher-level abstractions without encountering the limitations and data losses associated with representations manipulated by and stored in computer systems. The data repository approach recognizes human difficulties in dealing with the complexities of large systems. A data repository provides data storage and retrieval capabilities to assist a reverse engineer in collecting, recording, ordering, relating, interpreting, and recovering design information from a legacy system.

A Model On-Line Program

A randomly selected program from the Stock Control and Distribution (SC&D) system source code was used to develop a model of an on-line, CICS COBOL program (see Figure 20). The source code was reviewed to determine the size of each division and to isolate non-COBOL statements (i.e., CICS commands and database management system commands).

Program ZZLAI543, Program Stock List Changes Related Items, was originally written in 1989. There have been 66 subsequent versions of the program authored by 11 maintenance programmers. The program consists of 7,699 lines of source code, significantly larger than the average of 2,255 lines reported by Sneed and Jandrasics (1987). The Identification Division contains 878 lines; of these, 871 lines were notes related to program maintenance changes. Maintenance notes are relatively cryptic and provide limited information relative to program changes.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  ZZXXXXNNN.
```

```
ENVIRONMENT DIVISION.
DATACOM SECTION.
      MONITOR IS CICS
      ID-AREA IS ID-AREA-IDEATE
      PRINT GEN.
```

```
DATA DIVISION.
WORKING STORAGE SECTION.
      Flags, control records, transactions, input records
DATA-VIEW DVDF203U
      PREFIX IS DIFU-
      ACCESS KEY IS NIIN-KEY.
LINKAGE SECTION.
      Transaction Work Area (TWA), parameters.
```

```
PROCEDURE DIVISION.
 EXEC CICS HANDLE
      ABEND
      LABEL (9999-CICS-ABEND)
 END-EXEC.

 EXEC CICS ADDRESS TWA (BLL-TWA)
      PERFORM INITIALIZE-SEC THRU EXIT
      PERFORM PROCESS-TRANS THRU EXIT
 END-EXEC.

 EXEC CICS RETURN
 END-EXEC.
 GOBACK.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PERFORMED PARAGRAPHS.
 READ AND HOLD XYZ WHERE KEY EQUAL value.
 FREE LAST XYZ.
 READ AND HOLD NEXT XYZ.
 UPDATE XYZ.
 DELETE XYZ.
 WRITE XYZ.

 EXEC CICS LINK PROGRAM ('ZZXXXNNN')
      COMMAREA (parms)
      LENGTH (nnn)
 END-EXEC.
```

Figure 20. Model CICS on-line program.

The Environment Division consists of 11 lines and contains a Datacom Section identifying CICS as the monitor. As expected for an on-line program, no files are used by the program.

The Data Division consists of 1,445 lines, slightly more than the average 1,100 lines reported by Sneed and Jandrasics (1987). Much of the working storage space is set up for various transactions and records generated during program processing. Database tables accessed by the program were described in the working storage section as Dataview statements (DVxxxnnx); 31 tables are identified for program access. Also included in the Data Division is a short Linkage Section used to define a Transaction Work Area (TWA) and three parameter fields for use by calling and called programs.

The Procedure Division consists of 5,336 lines of code, substantially larger than the average of 2,250 lines. COBOL input/output verbs do not work under CICS; therefore, CICS commands are used (Lim, 1986). Several CICS execute commands are included in the Procedure Division; one of these is the LINK PROGRAM ('xxxxxnnn') command used to pass control to another program. Database access commands (e.g., Read and Hold, Update, Write, Delete, and Free) are numerous. There are few comments or notations in the Procedure Division.

The large size of the Data and the Procedure Divisions and the number of database tables accessed suggests complicated program logic and multiple transaction types.

A Model Batch Program

A COBOL program using multiple files was selected to develop a model of a batch program (see Figure 21). The source code was reviewed to identify division sizes and to identify unique Datacom/DB database commands and accessed tables.

Program ZZLAD058, Extract Transaction Data for Interface Systems, was written in 1986. There have been 55 subsequent versions of the program authored by 16 maintenance programmers. The program consists of 3,239 lines of source code, much larger than the average of 2,255 lines reported by Sneed and Jandrasics (1987).

The Identification Division consists of 204 lines; 195 lines are notes related to program maintenance changes. Maintenance notes are relatively detailed and indicate how particular parts of the program have been changed.

The Environment Division consists of 36 lines and includes 27 file select statements.

The Data Division consists of 778 lines, about 300 lines less than the average reported by Sneed and Jandrasics (1987). Only four database tables are identified in Dataview statements.

The Procedure Division consists of 2,219 lines, close to the average size of 2,250 lines reported by Sneed and Jandrasics (1987). The Procedure Division has several major

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  ZZXXXXNNN.
--------------------------------------------------------------
ENVIRONMENT DIVISION.
DATACOM SECTION.
      ID-AREA IS ID-AREA-IDENT.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM370.
OBJECT-COMPUTER. IBM370.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT A3543B0    ASSIGN TO UT-S-A353B0U.
--------------------------------------------------------------
DATA DIVISION.
FILE SECTION.
FD A353B0
   LABEL RECORDS ARE STANDARD
   RECORDING MODE IS F
   RECORD CONTAINS 90 CHARACTERS
   BLOCK CONTAINS 0 RECORDS
   DATA RECORD IS 53B0-D009A-D1.
01 53B0-D009A-DI    PIC X(90).

WORKING STORAGE SECTION.
      Flags, control records, transactions, input records
DATA-VIEW DVINF01U
      DATADICTIONARY NAME IS DVINF01U
      PREFIX IX INF-.
DATA-VIEW DVCTF02R
      ACCESS KEY IS ‘value’.
--------------------------------------------------------------
PROCEDURE DIVISION.
 ENTER-DATACOM-DB.
0000-MAINLINE.
 PERFORM 1000-ACCEPT-CNTRL-REC THRU 1000-EXIT.
 PERFORM 1100-PROCESS-CNTRL THRU 1100-EXIT.
 PERFORM 8500-DISP-TOTALS THRU 8500 EXIT.
 CALL ‘SUBPROG’ USING X, Y, Z.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PERFORMED PARAGRAPHS.
 READ DVCTF02R WHERE value = value.
 FOR EACH DVINF01U
     WHERE (value = value)
     HOLD RECORD.
 WHEN END.
 WHEN ERROR.
```

Figure 21. Model COBOL batch program.

components and primary processing is based on daily, weekly, biweekly, monthly, and as required processing as determined by a control record input. There are a total of 51 open-for-output statements for 27 files.

A Model Fourth-Generation Language (4GL) Program

A randomly selected IDEAL application was used to develop a model of a fourth-generation language (4GL) program (see Figure 22). IDEAL, a component of the Datacom/DB database management system, is an easy-to-use application program generator designed to facilitate access to database tables. IDEAL program structure resembles COBOL, but is less wordy. IDEAL is most often used for on-line programs. The various parts of an IDEAL program are similar to the four main divisions of a COBOL program. Line numbers are not used; source code lines are estimated.

Program ZZLAI304, Route D035A On-line Transactions, was originally written in 1988. There have been 39 versions prepared by 14 authors. The program consists of 1,340 lines of source code. A direct comparison with average COBOL program size is not possible because of language differences. A remarks section contains 300 lines explaining the maintenance changes.

The Program Section contains about 80 lines of source code and a short description of the program function. The program was created to route on-line transactions received by a

```
->PROGRAM ZZXXXXNNN
STATUS PROD IDEAL
DATE CREATED
DATE MODIFIED
DATA COMPILED
RUN STATUS PRIVATE
LANGUAGE IDEAL
SHORT-DESC 'text'
TEXT1 'text'
USES-DATAVIEW DVITF12R
USES-PROGRAM ZZXXXNNN
```

```
->WORKING DATA
  1 WS-PARM-3
    2 WS-MSG-ID      X 4
    2 W-MSG-DESC   X 72
```

```
->PARAMETER DATA
   1 PARM-1               UI
     2 PARM-DATE       X 8
     2 JULI-DATE        X 5
   1 PARM-2  X 200     UI
```

```
->PROCEDURE DATA
<<MAIN>> PROCEDURE
   SELECT
       WHEN WS-FIL-ID = 'value'
          DO P804-ZZXXXNNN
       WHEN NONE
          processing
    END SELECT

    SET parameters
ENDPROC
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
<<P804-ZZXXXNNN>> PROCEDURE

   SET BEFORE-AFTER = 'BEFORE'
   SET CALL-PROGRAM = 'ZZXXXNNN'
   CALL ZZXXXNNN USING parameters
   RELEASE PROGRAM ZZXXXNNN
   SET BEFORE-AFTER = 'AFTER'
ENDPROC
END-PROGRAM
```

Figure 22. Model 4GL program.

communications application to the proper processing program based on file identification and transaction type. This section identified one database table and 44 called programs.

The Working Data Section contains 60 lines of source code and establishes temporary working storage locations.

The Parameter Data Section contains 11 lines of source code. This section is equivalent to the Linkage Section in a COBOL program and is used to communicate between calling and called programs.

The Procedure Section contains 1,000 lines of source code. This section consists of a Main Procedure and a series of subroutines or called procedures.

Reverse Engineering Process Output Products

The reverse engineering methodology was developed to recover functional design information from legacy system source code to support the preparation of a replacement system functional description. General requirements for the information in the methodology output products were:

1. The functions implemented in the legacy system should be described in technology-independent form. These descriptions should be neutral with respect to who performs a function and how the function is performed.

2. The functional description should be in narrative form in non-technical terms to facilitate review and validation by domain area specialists.
3. The functions should be presented in a hierarchical structure to facilitate activity grouping and to allow the structure to be reviewed from several levels.
4. A conceptual data structure in the form of an entity-relationship diagram should be included.
5. A data dictionary defining domain objects, terms, and attributes used in the functional process and the conceptual data models should be included.
6. A graphic model of the functional processes capable of displaying data sources, data destinations, internal data flows, external data flows, process relationships, and major data stores.
7. A description of each input interface from an external system. The narrative should be detailed enough to capture the functional requirements of the interface.
8. A description of each output interface to an external system. The narrative should be detailed enough to capture the functional requirements of the interface.
9. An electronic repository to capture primitive, intermediate and high level results of the reverse engineering effort to support verification and additional analysis should be constructed to support manual efforts.

Within the Department of Defense, Military Standard 498 (MIL-STD-498, 1994) establishes requirements for documentation of military software systems. The format and

content requirements for documents are contained in data item description (DID) attachments.

Two document types are appropriate for describing data recovered by reverse engineering: an Operational Concept Document (OCD) and a Database Design Description (DBDD).

The OCD requirements are described in the DID DI-IPSC-81430 attachment. Specific content requirements for the OCD are delineated in paragraph 10.2, Content Requirements. The portion of the contents pertinent to reverse engineering design information recovery is (OCD) paragraph 5.3, Description of the New or Modified System, specifically subparagraphs b through e:

> b. Major system components and the interconnections among these components.
>
> c. Interfaces to external systems or procedures.
>
> d. Capabilities/functions of the new or modified system.
>
> e. Charts and accompanying descriptions depicting inputs, outputs, data flow, and manual and automated processes sufficient to understand the new or modified system or situation from the user's point of view. (p. 5)

The DBDD requirements are specified in the DID DI-IPSC-81437 attachment. Specific content requirements for the DBDD are delineated in paragraph 10.2, Content Requirements. The portion of the contents pertinent to reverse engineering design information recovery is (DBDD) paragraph 4, Detailed Design of the Database:

> This section shall be divided into paragraphs as needed to describe the detailed design of the database. The number of levels of design and the names of those

> levels shall be based on the design methodology used. Examples of database design levels include conceptual, internal, logical, and physical. (p. 4)

Subparagraphs of paragraph 4, Name of Database Design Level, delineate the following content:

> This paragraph shall identify a database design level and shall describe the data elements and data element assemblies of the database in the terminology of the selected design method. The information shall include the following, as applicable, presented in any order suited to the information to be provided:
>
> a. Characteristics of individual data elements in the database design.
>
> b. Characteristics of data element assemblies (records, messages, files, arrays, displays, reports, etc. in the database design. (p. 5)

Although this research concentrated on process reverse engineering as opposed to data reverse engineering, both documentation requirements were identified to clearly reflect the additional work required to develop a comprehensive reverse engineering methodology.

Two products were proposed to meet the requirements for the OCD: a conceptual process model and a visual process model (Miller, 1995a). The conceptual process model is presented in a hierarchy of key areas, tasks, subtasks, and activities representing the functional processes implemented in a legacy system. Each primitive (bottom-level) activity is described in narrative format in application domain terminology. Descriptions are abstracted to ensure no implementation-specific details are retained in the narrative and are neutral with respect to who performs the activities. The narrative descriptions are supported and clarified by the visual process model.

The visual process model is based on traditional data flow diagrams (DFD). DFD complement the narrative description by displaying information (e.g., sources and destinations of data flows) that is not easily represented in narrative format. The DFD approach can also be used as a diagramming tool to capture low-level program information. Programming information can then be abstracted into higher-level conceptual process models.

The entity-relationship diagram (ERD) approach is the recommended modeling tool for data reverse engineering. Available methodologies and tools for data reverse engineering are described by Aiken (1996).

Interfaces with external systems (both input and output) were modeled as functions in the conceptual process model. Each input or output file is associated with a separate activity (i.e., separate activities for daily and monthly versions of an input or output interface file) in the narrative and visual versions of the process model.

An electronic repository to store reverse engineering information was proposed as a tool in managing the large volume of information in the application of the methodology. Although not a product of the methodology, such a repository aids in the production of required output products. The requirements for the repository were defined during methodology development in the next section.

Developing the Reverse Engineering Methodology

Purpose

The purpose of the reverse engineering methodology is to define and describe techniques, processes, and procedures to aid in the recovery of design information from a legacy system. Recovered design information is used to support the creation of a replacement system functional description.

Scope

The methodology focused on the military logistics system domain, specifically legacy systems written in COBOL and operating on IBM mainframe computers with an MVS/ESA operating system. The database management system used in the target environment is CA-Datacom/DB.

At the high level, a complete methodology, including data structure recovery,was described. The emphasis at the low level was on the recovery of design information from processes embedded in source code.

A major premise was that reverse engineering is a component of software engineering and subject to the same rigor and discipline as any other software engineering component. The information system (reverse engineering methodology) to be designed was envisioned as a manual system; good development practice demands that requirements and preliminary system design be implementation independent. Thus, the ultimate

implementation of the system was not a consideration during requirements analysis and specification. The physical implementation phase normally associated with instituting a system on a computer was omitted. The conceptual, definition, and logical phases provide sufficient information to manually apply the methodology.

Strategy

Several important points relative to reverse engineering were clear from the literature review and drove the strategy behind the reverse engineering methodology. Process reverse engineering is a human activity. Efforts to develop computer-based tools to recover design information from source code have been successful only with small programs in a research environment. The intelligence required for reverse engineering must be provided by specialists.

Source code alone is insufficient to recover design information. In the forward engineering process, there is a considerable loss of domain knowledge when requirements are mapped to physical system implementation. Lost domain knowledge must be provided from external sources. Documentation does not contain sufficient domain knowledge to support reverse engineering. Domain specialists must support any reverse engineering project. Aiken (1996) estimated participation by functional and technical personnel in data reverse engineering projects can greatly reduce the amount of time and resources required.

The number of legacy systems in need of reverse engineering is staggering and continues to grow. A reverse engineering methodology must support rapid abstraction to a level higher than source code.

Computer-based tools are needed to support the collection and management of recovered design information. The volume of data to be manipulated during reverse engineering is too massive to be manually accomplished.

Both data and process reverse engineering are necessary. There is an inseparable link between data and processes. A complete reverse engineering methodology must address both areas. Data reverse engineering is less complicated than process reverse engineering. There are a number of available data reverse engineering tools capable of creating database schema from existing COBOL files and data division structures. However, automated tools do not recover knowledge of the data structures. Reverse engineers must study and understand the recovered structure before true reverse engineering takes place. Anything less is transformation rather than reverse engineering.

A major hypothesis in this investigation was that it is possible to reverse engineer an unstructured system into a hierarchical structure. This hypothesis is based on the observation that the lack of structure in a system is found in program code, i.e., the system is made up of a group of programs that bear little direct correlation to the structure of the problem. Maintenance difficulties associated with legacy systems are caused, at least in

part, because the correspondence between problem structure and program structure is difficult to identify.

By identifying functions implemented in code and separating the functions from the implementation environment, a reverse engineer can create a hierarchical structure suitable for forward engineering a replacement system. A forward engineering model based on hierarchical abstraction is therefore considered appropriate for the purpose of supporting understanding of a reverse engineering model.

In reverse engineering, there are two areas of concern with respect to software: (a) the internal syntax and semantics of a program, and (b) the external interfaces with the program environment. The internal program area is the most complicated of the two areas and is the heart of the reverse engineering problem.

External interfaces are relatively simple to identify as only a limited number of forms are possible. As shown in Figure 23, there is one input-only possibility (conceptually a terminal, but actually a keyboard), one output-only possibility (screen display), and two create-read-update-delete possibilities (files and database tables). The remaining consideration is how system components (programs or modules) interact with each other.

System component links (caller/called relationships) exist in two forms and two degrees.

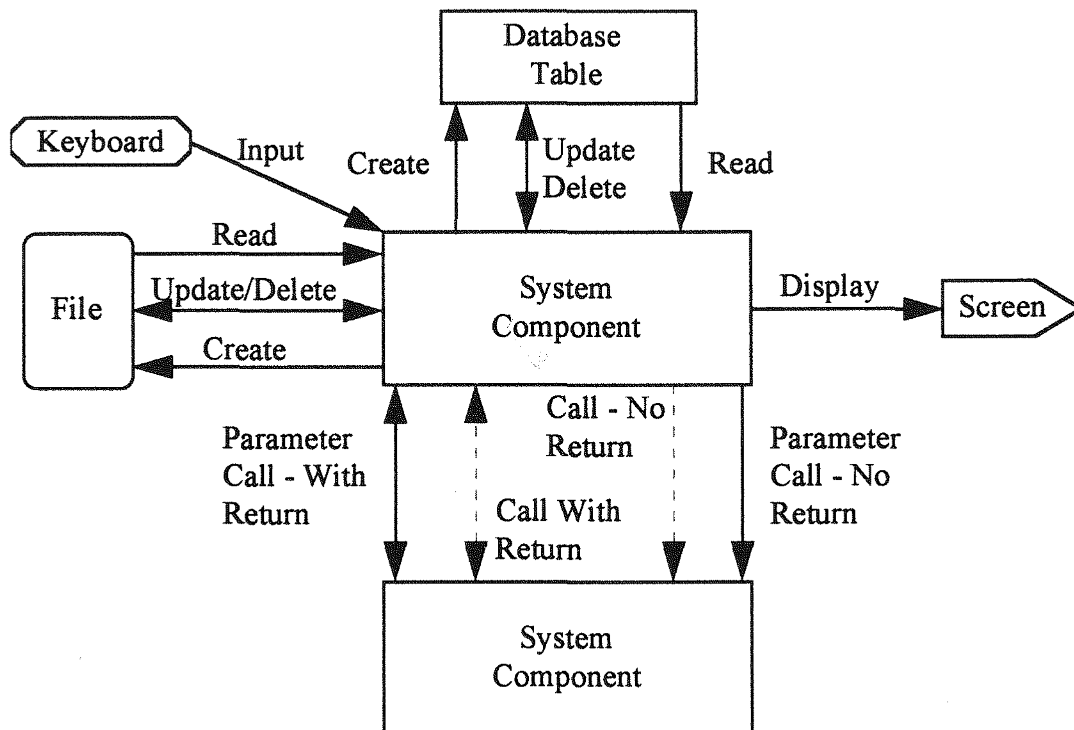


Figure 23. External interfaces of a system component.

The two forms are a call to a sub-component and a return to caller (represented as a double-headed arrow), and a call to a sub-component with no return to caller (represented as a single headed arrow). The two degrees are a weak call (no data is passed) represented by a dashed line, and a strong call (data is passed) represented by a solid line.

A reverse engineering methodology must be capable of capturing these interactions from source code and storing them in a format that supports the interpretation and abstraction of the internal program syntax and semantics.

Goals

Specific goals of the reverse engineering methodology were:

1. Easy to use, meaningful, and complete (Kaposi & Pyle, 1993).
2. Teachable and repeatable.
3. Practical and usable rather than theoretical.
4. Based on detailed knowledge and understanding of the nature of software and its forward engineering development.
5. Create a flexible framework to accommodate software systems of varying size, age, complexity, and criticality.
6. Clearly specify the tools and techniques to be used and the problems they are designed to work on (Canfora, et al., 1994).
7. Develop a representation with minimal restriction on the possible abstractions of information recovered, and not restrictive to a particular form of technical documentation (Grumman & Welch, 1992).
8. Identify operations to be carried out, inputs to be used, and outputs to be produced (Benedusi, Cimitile, & de Carlini, 1989).
9. Provide a data store to accommodate the quantity and complexity of data involved (Ostrolenk, et al, 1993).
10. Support complex problems by making multiple passes over the data, doing something simple in each pass (Orr, 1981).
11. Allow incremental reverse engineering by processing different system components at different times.

12. Apply abstraction to reduce the volume of data to be manipulated.
13. Allow a reverse engineer to identify important abstractions using insight, knowledge of the application domain, and knowledge of systems design (Bennett, 1993).
14. Allow for information solicitation from system functional users.
15. Allow domain specialists to provide missing domain knowledge to augment recovered design information.
16. Allow for verification of recovered information by domain specialists.
17. Describe recovered functions in non-technical narratives easily understood by functional users.
18. Augment the functional narrative descriptions with a visual model showing components, their dependencies and interactions, major sources and destinations of inputs and outputs, and conceptual data stores.
19. Recover conceptual constructs rather than mathematical representations (Mays, 1994).
20. Identify the purpose (telos) of software constructs in terms of application domain concepts (Karakostas, 1990).
21. Produce readable and understandable system models (McLaughlin, Estdale, & Tobin, 1993).

The Conceptual Process Model

The reverse engineering conceptual process model was presented as a hierarchical decomposition of key areas (single-digit numbers), tasks (two-digit numbers), and activities (three-digit numbers). The key areas, tasks, and activities represent a leveled

modeling approach to control the decomposition complexity. At each level in the model there are seven components (plus or minus two). Primitive elements, normally activities, are described in narrative form. Only the key areas are presented here; the complete decomposition hierarchy is presented in Appendix B. The narrative conceptual model was augmented by a visual process model. The visual process model uses traditional data flow diagrams to graphically represent the key areas, tasks, and activities described in the conceptual process model and to identify conceptual data stores. Each numbered element in the narrative process model is represented in a data flow diagram process. The complete visual process model is also located in Appendix B.

The context diagram establishes boundaries around the reverse engineering methodology (see Figure 24). The methodology is represented by a single process (a circle at the center of the diagram); major external entities (squares on the left and right of the diagram) provide data to the process or receive data from the process; data flows (directed, named arrows) represent the high-level flow of information into and out of the model. A data store (rectangle) indicates the necessity to temporarily hold some data while it is not flowing through the model. The symbols used in the context diagram are used in all lower-level decomposition diagrams.

The level 0 diagram shows how the key areas interrelate in the first level below the context diagram (see Figure 25). The seven key areas of the reverse engineering methodology are briefly described here.

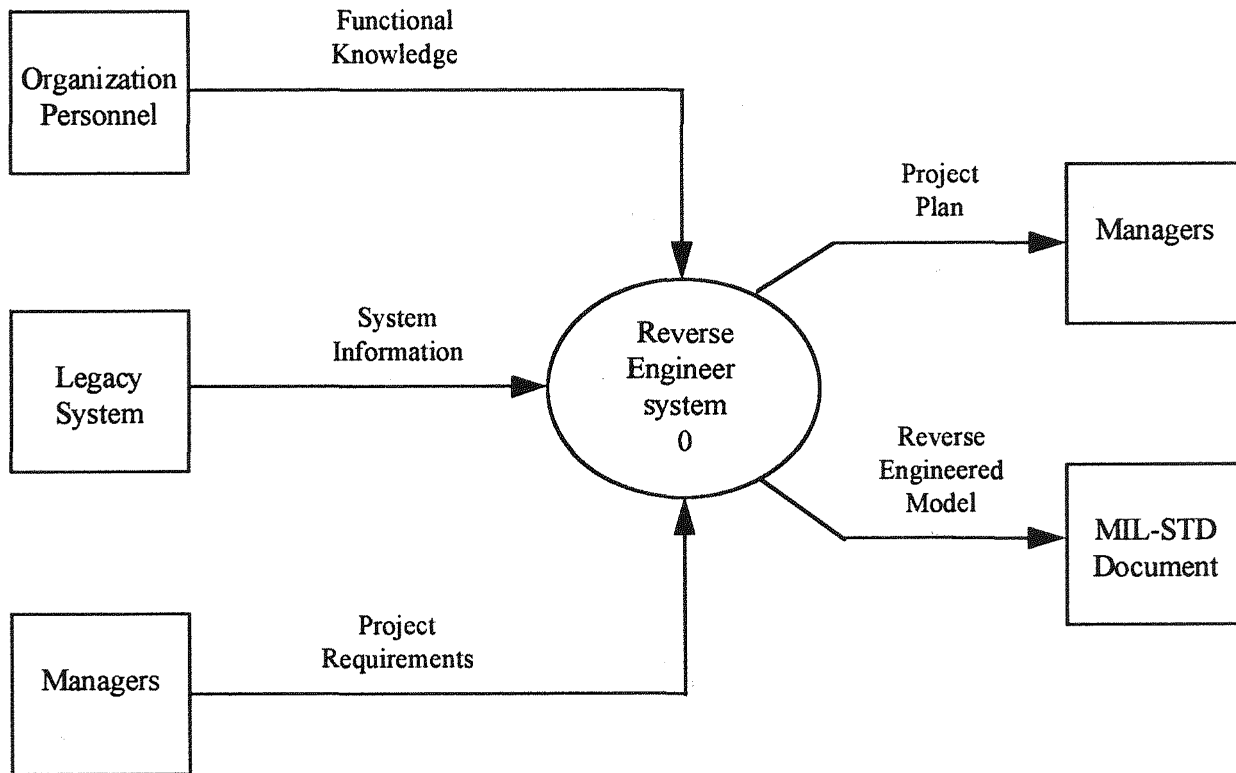


Figure 24. Context diagram.

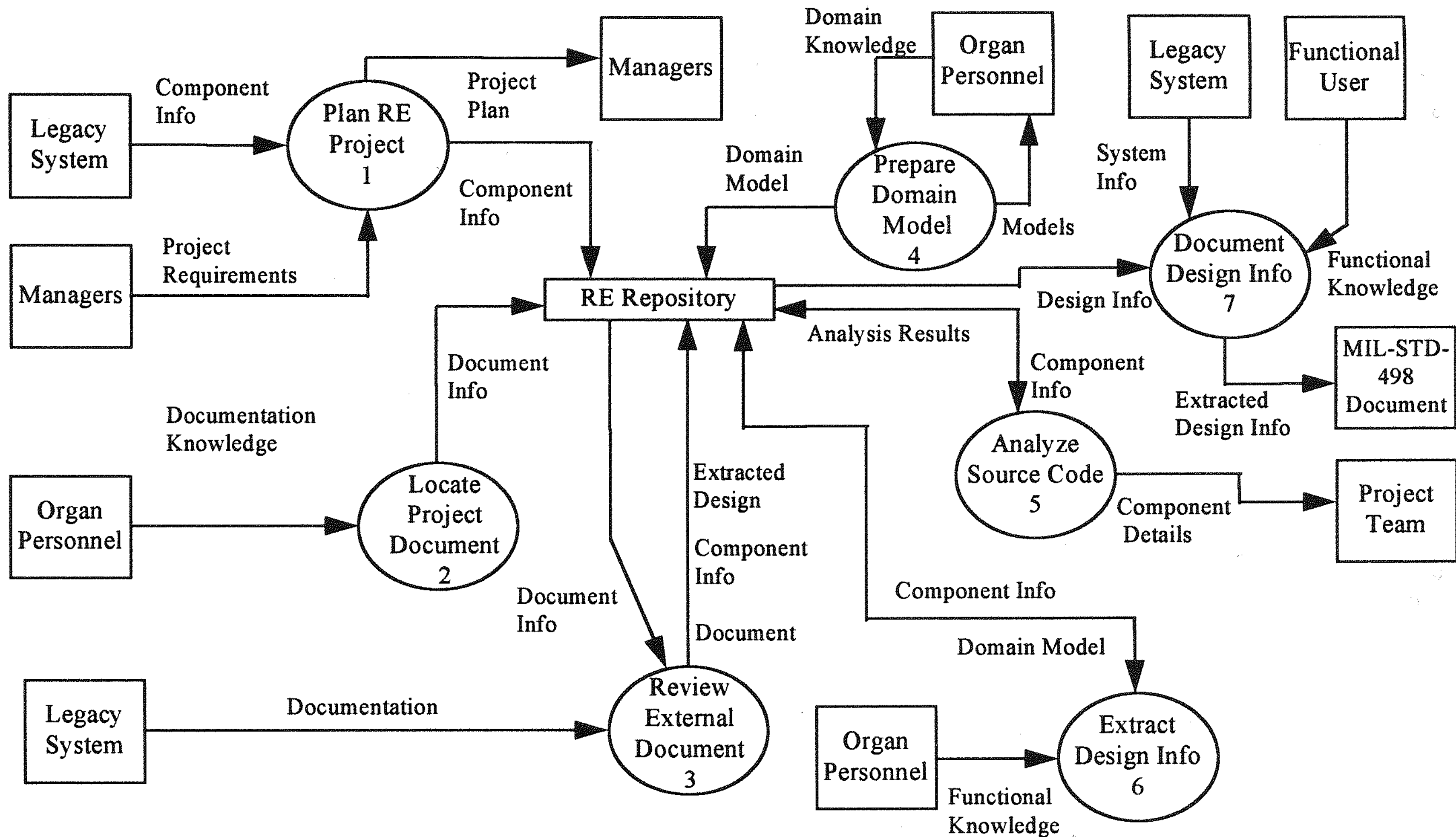


Figure 25. Level 0 diagram.

Key Area 1. Plan Reverse Engineering Project - This functional area results in the creation of a project plan specifying the scope, objectives, resource estimates, and schedule for completing a specific reverse engineering project.

Key Area 2. Locate Project Documentation - Legacy system documentation, even if it is outdated, incomplete, or incorrect, is useful as a source of domain information and for understanding original intent. This functional area results in the location of available external documentation.

Key Area 3. Review External Documentation - This functional area results in the preliminary assessment of documentation usefulness. Available external documentation is collected and cataloged to support the reverse engineering effort. Useful documentation is extracted and provided to the reverse engineering team.

Key Area 4. Prepare Domain Model - Domain knowledge represents the majority of missing source code information. This functional area results in the creation of a narrative description of the functional hierarchy perceived by domain specialists and functional users as being implemented in the legacy system. The resultant model serves as a target structure where recovered design information is placed.

Key Area 5. Analyze Source Code - The main focus of the reverse engineering methodology is to analyze legacy system source code to extract design information. This

functional area stresses the methodical review and collection of information about programs to permit modeling without further reference to source code.

Key Area 6. Extract Design Information - This functional area represents the interpretation of available program information to form the preliminary extracted design information model. Intuition, previous software engineering experience, deductive reasoning, causal connectivity, semantic concept formation, and fuzzy logic techniques may be applied to complete the key area. Domain specialists and functional users are consulted to aid in model development.

Key Area 7. Document Design Information - This functional area results in the delivery of the reverse engineering product. The recovered design information is consolidated into a preliminary model using the domain model as a framework. The preliminary model is verified by domain specialists and functional users, refined as necessary, and produced as a final model according to the deliverable format specified in the project plan.

The Conceptual Data Model

The conceptual data model, represented as an entity-relationship diagram (ERD) in Figure 26, shows the high-level data structure required to support the reverse engineering processes described in the conceptual process model. Entities were identified from narrative process descriptions and visual process model data stores. Relationships were identified by determining associations between entities.

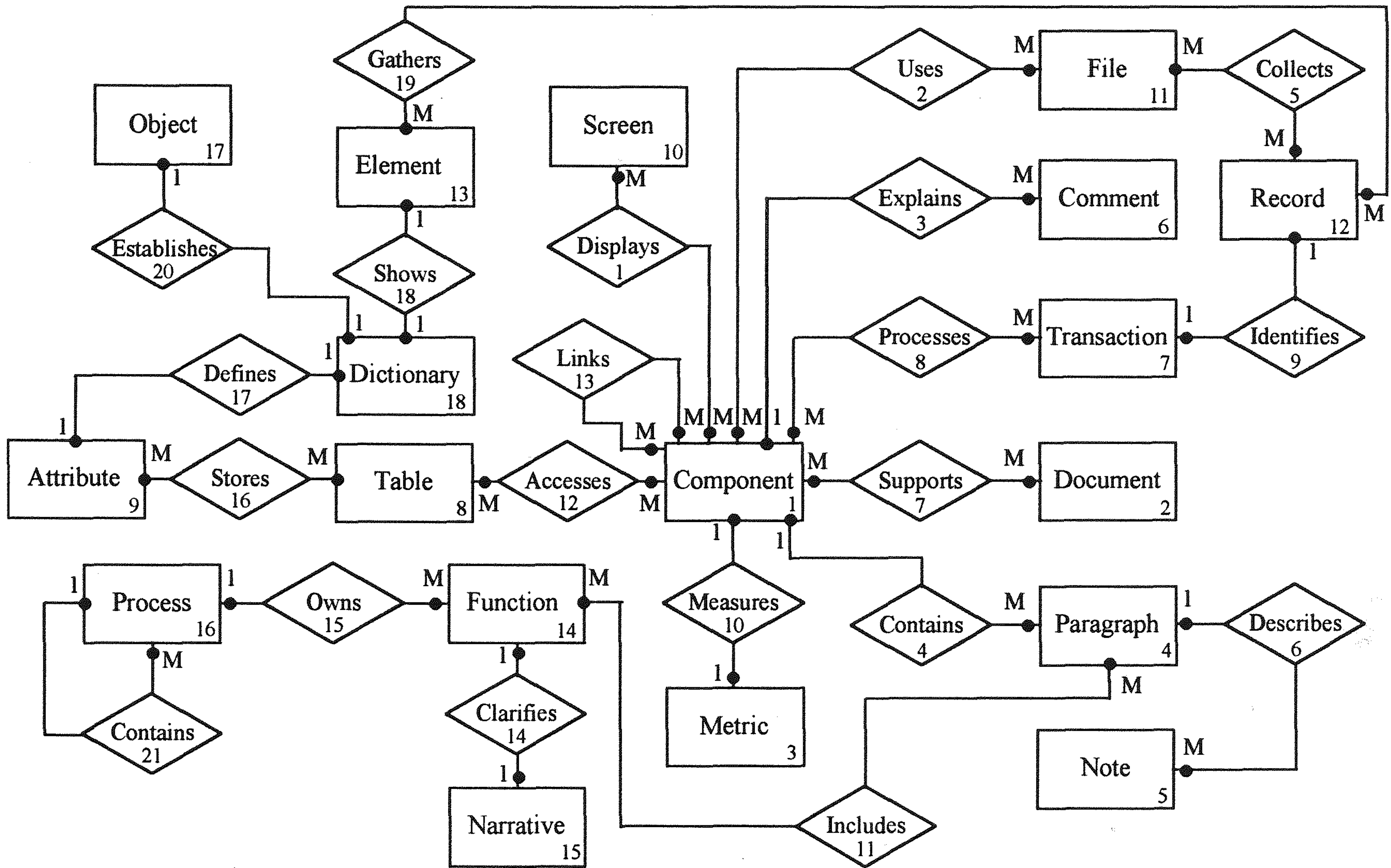


Figure 26. Conceptual data model.

The ERD consists of entities (rectangles) describing objects from the legacy system about which data is to be stored, and relationships (diamonds) showing associations between the entities. Entities are identified with singular nouns; relationships are identified with verbs. Each relationship is annotated as to its membership class (obligatory or non-obligatory) and membership degree (one to one, one to many, or many to many).

Obligatory membership is shown with a black dot on the intersection of the entity box and relationship line. Non-obligatory membership is shown with a dot on the relationship line outside the entity. A one-to-one relationship is shown with a 1 next to both entities connected by a relationship. A one-to-many relationship is shown with a 1 next to one entity and an M next to the other entity in a related pair. A many-to-many relationship is indicated by an M next to each of the related entities.

A separate numbering scheme is used to identify each entity and relationship. Entities are defined in Table 10, and relationships are defined in Table 11.

Table 10
Conceptual Data Model Entity List

Entity number	Entity name	Entity identifier	Description(s)
1	Component	ID-Component	A COBOL program, subroutine, or 4GL routine making up a system.
2	Document	No-Seq-Doc	A formal or informal publication describing a system component.
3	Metric	No-Seq-Metric	Estimated and actual time required to reverse engineer a system component.
4	Paragraph	No-Seq-Para	A subdivision of a system component consisting of a name and a series of statements identified by a (starting) line number.
5	Note	No-Seq-Note	In-line text describing the purpose or function performed by a paragraph (type I). Text written by a reverse engineer to describe a component paragraph (type A).
6	Comment	No-Seq-Comment	Component text describing a program's purpose or general processing; usually found in COBOL Remarks section or IDEAL program headers (type I). Text written by a reverse engineer to explain a system component (type A).
7	Transaction	Doc-ID-Code	A transaction identified by a common document identifier code (DIC).
8	Table	No-Seq-Table	A data structure associated with a database management system.
9	Attribute	No-Seq-Attribute	A column in a database table or a characteristic of an object represented by a table. Roughly equivalent to a data element.

Table 10. (continued)

Entity number	Entity name	Entity identifier	Description(s)s
10	Screen	ID-Screen	A terminal display generated by a component for data input and output.
11	File	ID-File	A collection of data records not associated with a database management system. May be internal or external to the system.
12	Record	No-Seq-Record	A data structure associated with a file and made up of data elements.
13	Element	No-Seq-Element	A data field associated with a file record. May be a structure.
14	Function	No-Seq-Function	The name of a process performed by a component. Established by a reverse engineer as a result of interpreting a component paragraph.
15	Narrative	No-Seq-Narrative	The natural language description of a function performed by a component paragraph.
16	Process	No-Seq-Process	A hierarchical component of a domain model representing the general functions implemented in a system. Consists of key areas, tasks, subtasks, and activities in descending order. Provides the framework for organizing functions into the reverse engineered function model.
17	Object	No-Seq-Object	Domain elements acted upon by the domain model processes. May become data entities when a reverse engineered function model is forward engineered.
18	Dictionary	No-Seq-Entry	A repository for storing definitions of acronyms, attributes, and data elements and descriptions of domain objects.

Table 11
Conceptual Data Model Relationship List

Relationship number	Relationship name	Associated entities	Description
1	Displays	Component Screen	Associates a terminal screen with the component displaying it.
2	Uses	Component Files	Identifies the files used by a system component and identifies file activity (i.e., read, write, update).
3	Explains	Component Comment	Associates descriptive comments with a particular component.
4	Contains	Component Paragraph	Associates extracted paragraph names with the source component.
5	Collects	File Record	Associates record formats with the files they are used in.
6	Describes	Paragraph Note	Relates descriptive notes with a particular component paragraph.
7	Supports	Component Document	Associates specific life-cycle documents with a component.
8	Processes	Component Transaction	Associates transactions with a particular component.
9	Identifies	Transaction Record	Associates a transaction with a specific record format.
10	Measures	Component Metric	Associates a component with the specific metric values used to measure the component.
11	Includes	Paragraph Function	Identifies one or more functions extracted from a paragraph.
12	Accesses	Component Table	Identifies the database tables accessed by a component.
13	Links	Component Component	A recursive relationship associating a component with its subordinate called components.
14	Clarifies	Function Narrative	Links descriptive text with a reverse engineered function title.
15	Owns	Function Process	Relates a reverse engineered function to a specific parent activity in the domain model process.
16	Stores	Table Attribute	Identifies the specific attributes contained in a database table.
17	Defines	Dictionary Attribute	Associates an attribute with its definition in the dictionary.

Table 11. (continued)

Relationship number	Relationship name	Associated entities	Description
18	Shows	Dictionary Element	Associates a data element with its definition in the dictionary.
19	Gathers	Record Element	Relates a data element to the record it appears in.
20	Establishes	Dictionary Object	Associates a domain object with a dictionary entry that explains it.
21	Contains	Process Process	A recursive relationship showing the hierarchical structure of the domain model.

The Definitional Model

The definitional process model is normally used to add timing, frequency, and location information to activities. This step was omitted as the objective is to define a manual methodology.

The definitional data model or high-level conceptual data model (the ERD) was expanded into a detailed definitional model by applying relational table formation rules to the entity relationships (Howe, 1983). One entity, Dictionary (E-18), and five relationships (R-9, R-10, R-17, R-18, and R-20) did not result in table formation. The list of skeletal tables formed is shown in Table 12.

Table 12
Skeletal Table List

Table number	Table name	Entity (E-n) or Relationship number (R-n)	Identifier (P = Primary, F = Foreign)
1	Acronym	Added	No-Seq-Acronym (P)
2	Attribute	E-9	No-Seq-Attribute (P)
3	Comment	E-6	No-Seq-Comment (P)
4	Component	E-1	ID-Component (P)
5	Component-Comment	R-3	ID-Component (F) No-Seq-Comment (F)
6	Component-Component	R-13	ID-Component-Calls (F) ID-Component-Called (F)
7	Component-Document	R-7	ID-Component (F) No-Seq-Doc (F)
8	Component-File	R-2	ID-Component (F) ID-File (F)
9	Component-Paragraph	R-4	ID-Component (F) No-Seq-Para (F)
10	Component-Screen	R-1	ID-Component (F) ID-Screen (F)
11	Component-Table	R-12	ID-Component (F) No-Seq-Table (F)
12	Component-Transaction	R-8	ID-Component (F) Doc-ID-Code (F)
13	Document	E-2	No-Seq-Doc (P)
14	Element	E-13	No-Seq-Element (P)
15	File	E-11	ID-File (P)
16	File-Record	R-5	ID-File (F) No-Seq-Record (F)
17	Function	E-14	No-Seq-Function (P)

Table 12. (continued)

Table number	Table name	Entity (E-n) or Relationship number (R-n)	Identifier (P = Primary, F = Foreign)
18	Metric	E-3	No-Seq-Metric (P)
19	Narrative	E-15	No-Seq-Narrative (P)
20	Narrative-Function	R-14	No-Seq-Narrative (F) No-Seq Function (F)
21	Note	E-5	No-Seq-Note (P)
22	Object	E-17	No-Seq-Object (P)
23	Paragraph	E-4	No-Seq-Para (P)
24	Paragraph-Function	R-11	No-Seq-Para (F) No-Seq-Function (F)
25	Paragraph-Note	R-6	No-Seq-Para (F) No-Seq-Note (F)
26	Process	E-16	No-Seq-Process (P)
27	Process-Function	R-15	No-Seq-Process (F) No-Seq-Function (F)
28	Process-Process	R-21	Is-Parent-Of (F) Is-Child-Of (F)
29	Record	E-12	No-Seq-Record (P)
30	Record-Element	R-19	No-Seq-Record (F) No-Seq-Element (F)
31	Screen	E-10	ID-Screen (P)
32	Table	E-8	No-Seq-Table (P)
33	Table-Attribute	R-16	No-Seq-Table (P)
34	Term	Added	No-Seq-Term (P)
35	Transaction	E-7	Doc-ID-Code (P)

The Logical Model

In the logical modeling phase, the conceptual process model and the definitional data model were further refined to produce the preliminary reverse engineering design. Logical modeling is the intermediate step between the functional concept of *what* is accomplished and the initial *how* it will be accomplished. One of the objectives of logical modeling is to identify the association between process and data. The output of this phase was a reverse engineering methodology that can be manually applied.

The logical process model validates the logical data model by showing where every table is created and read (and possibly updated and deleted). At the end of the modeling phase, unused tables reflect either missing processes or extraneous data. Processes not supported by a table indicate either missing data or extraneous processes. Tables accessed by a service are identified along with the key for the table. Table attributes used by a service are omitted for brevity, but are implied by descriptions of the services.

The Logical Process Model

Logical Modeling Technique

Logical process modeling was accomplished by applying service analysis (Davis & Shah, 1985). Service analysis identifies the input, output, and processing functions required in a system. Services describe unit tasks performed by individuals. The ultimate goal of service analysis is to develop a comprehensive requirements statement for use in the design phase. An abbreviated form of service analysis was used here.

Activities from the conceptual process model are analyzed to identify a series of services required to perform the activity. A service description or service profile is normally prepared to describe each service in detail. The degree of detail depends on how physical design is to be accomplished. For this application, the identification of manual services and a supporting data structure mark the end of analysis; the amount of detail required in the services is therefore limited. The services described herein, however, could be extended into the design phase to produce a computer-based tool to perform many of the services and to partially automate the remainder.

Services are identified by using the activity number and title from the conceptual data model (located in Appendix B) and listing a series of lettered services as subparagraphs. Many of the tasks from key area 7 are skipped because they deal with data reverse engineering. Activities are combined into a single series of services when they are associated with the same data.

Tables used in a service are identified by name (underlined) and number. Table identifiers are indicated by (I); foreign keys in a table are identified by (F). Note that associative entity tables (join tables) contain two foreign keys; the foreign keys comprise the identifier. Complete table descriptions including attributes are contained in Appendix C.

Logical Model Services

1.1.1 Identify Target System - 1.1.5 Identify Project Deliverable.

a. Review project requirements directive.

b. Determine identity of the system to be reverse engineered.

c. Determine scope of reverse engineering effort.

d. Identify constraints levied against effort (e.g., required completion data, number of personnel available, budget limitations, and functional users).

e. Determine final documents to be produced.

f. Clarify provisions of tasking directive with the issuing authority.

Tables/Identifiers: None.

1.2.1 Identify Run Unit.

a. Review batch operations documents.

b. Identify program work units.

c. Print work unit job control language (JCL).

Tables/Identifiers: None.

1.2.2 Identify Component.

a. Extract component (program) names from batch JCL.

b. Review CICS Processing Program Table (PPT).

c. Extract component name from DFHPPT macro statements.

d. Review CICS Program Control Table (PCT).

e. Extract TRANSID from DFHPCT macro statement.

Tables/Identifiers:

Component (4)
ID-Component (I)

1.2.3 Classify Component - 1.2.4 Determine Component Type.

a. Record component class: PR for program or SR for subroutine. Subroutine components normally have a PROCEDURE DIVISION USING statement.

b. Record component type: BA for batch or OL for on-line. On-line components can be identified by the MONITOR IS CICS statement or by the absence of input and output files.

Tables/Identifiers:

Component (4)
ID-Component (I)

1.2.5 Create Subsystem Structure.

a. Review operational documents.

b. Extract subsystem structure information.

c. Prepare subsystem structure diagram.

d. Store structure diagram in RE library.

Tables/Identifiers: None.

1.3.1 Copy Source Code - 1.3.5 Print Reference Listing.

a. Copy source text from mainframe system in ASCII text format.

b. Establish a directory structure for storing source code on a personal computer.

c. Load source text into individual subdirectories.

d. Print directory contents.

e. Store directory listings in RE library.

Tables/Identifiers: None.

1.4.1 Extract Descriptive Information - 1.4.4 Extract Linking Information.

a. Review component source listing.

b. Record descriptive information

c. Extract header comments (Type = I).

d. Write descriptive comments (Type = A).

e. Count input files.

f. Count output files.

g. Count input-output files.

h. Count display screens.

i. Count output reports.

j. Count number of programs called.

k. Record time required for initial component review.

Tables/Identifiers:

Component (4)
ID-Component (I)

Comment (3)
No-Seq-Comment (I)

Component-Comment (5)
ID-Component (F)
No-Seq-Comment (F)

Metric (18)
No-Seq-Metric (I)
No-Seq-Metric (F)

1.5.1 Assess Component Structure - 1.5.3 Assess Naming Conventions.

a. Review program source listing.

b. Count GO TO statements.

c. Count PERFORM statements.

d. Count REDEFINES statements.

e. Count number of paragraphs.

f. Sample paragraph length to calculate average paragraph size in lines of code.

g. Assign program structure rating (1 = good, 10 = poor).

h. Review in-line comments for number and clarity.

i. Assign comment rating (1 = good, 10 = poor).

j. Sample program names to calculate average size in number of characters.

k. Assign name rating (1 = good, 10 = poor).

Tables/Identifiers:

Component (4)
ID-Component (I)

1.5.4 Assign Complexity Index.

a. Retrieve component size values and ratings.

b. Compute complexity index as follows (round all decimal values to the nearest whole number):

Index-Complexity = ((No-Files-In) -2) x 10 +

(No-Files-Out -1) x 10 +

(No-Files-IO -1) x 10 +

(No-Screens -1) x 10 +

(No-Reports - 1) x 10 +

((No-Lines-Source - 2250)/100) +

No-Versions +

No-Authors +

((No-Data-Div-Lines - 1,100)/100) +

((No-Proced-Div-Lines - 1,200/100) +

No-Program-Called +

No-GOTO-Stmnts

((No-Para-Lines-Avg - 50)/10) +

Program-Structure-Rating +

Rating-Comments +

Rating-Names

Tables/Identifiers:

<u>Component</u> (4)
ID-Component (I)

1.6.1 Identify Domain Specialist - 1.6.2 Identify Functional Technician.

a. Interview functional managers, operations personnel, and maintenance personnel.

b. Identify domain specialist point of contact for each program component.

c. Identify functional technician point of contact for each program component.

Tables/Identifiers:

<u>Component</u> (4)
ID-Component (I)

1.6.3 Estimate Personnel Required.

a. Review component summary information (number of programs, languages, sizes).

b. Assess application domain complexity.

c. Estimate time required for organizational personnel to prepare domain model.

d. Estimate time required for organization personnel to assist in code review.

e. Provide estimate of personnel required for project plan.

Tables/Identifiers:

Component (4)
ID-Component (I)

1.7.1 Review Component.

a. Prepare list of components by size and complexity index.

b. If metrics are available, use past experience to estimate person-hours required to reverse engineer each component.

c. If metrics are not available, make best estimate of time required to reverse engineer each component.

Tables/Identifiers:

Component (4)
ID-Component (I)
No-Seq-Metric (F)

Metric (18)
No-Seq-Metric (I)

1.7.2 Consolidate Resource Projection.

a. Summarize initial estimates of reverse engineering analysis time required by component.

b. Divide estimated total time in hours by number of personnel available to determine calendar time required.

c. Divide estimated total time in hours by number of days (or weeks, months) to determine number of reverse engineers required.

Tables/Identifiers:

Component (4)	Metric (18)
ID-Component (I)	No-Seq-Metric (I)
No-Seq-Metric (F)	

1.7.3 Prepare Work Schedule - 1.7.4 Write Project Plan.

a. Using the consolidated projection and available resources, prepare a work schedule showing the sequence of activities and projected completion dates for each component.

b. Using the completed work schedule and target system description, objectives, scope, deliverable format, and constraint information from the tasking directive, prepare a reverse engineering project plan.

c. Distribute plan to appropriate managers and members of the project team.

Tables/Identifiers: None.

2.1 Identify Requirements Documentation - 2.9 Prepare Document List.

a. Interview configuration, operations, technical, and functional personnel to identify relevant system documentation.

b. Prepare a preliminary list of available documentation and locations.

c. Update document information.

Tables/Identifiers:

<u>Document</u> (13)
No-Seq-Doc (I)

3.1 Collect Document - 3.2 Catalog Document.

a. Locate each document on the list.

b. Copy document if original is not available.

c. Catalog document and file in the RE library.

d. Update the document index.

Tables/Identifiers:

<u>Document</u> (13)
No-Seq-Doc (I)

3.3 Evaluate Document.

a. Determine date written, date last changed, requiring directive, and number of pages.

b. Evaluate document for relative worth in supporting the reverse engineering effort in regards to number of changes, content, and clarity of writing.

c. Assign a document usefulness rating (1 = poor, 10 = high).

d. Prepare a short document evaluation narrative for the document.

Tables/Identifiers:

<u>Document</u> (13)
No-Seq-Doc (I)

3.4 Identify Missing Document.

a. Prepare a list of documentation not available or not found.

b. Make a "not available" entry for a missing document or for a document not prepared during system development in the RE library.

c. Notify managers of missing documentation.

d. Request managers provide informal documentation to replace missing information.

Tables/Identifiers:

<u>Document</u> (13)
No-Seq-Doc (I)

<u>Component-Document</u> (7)
ID-Component (F)
No-Seq-Doc (F)

3.5 Locate Traceability Matrix.

a. Review available documentation to locate requirements traceability information (links requirements elements with preliminary design elements and preliminary design elements with physical program components).

b. Review operational systems related to the target system (the traceability matrix may be implemented in an automated system).

c. If found, store the traceability matrix in the RE library.

Tables/Identifiers: None.

3.6.1 Review Functional Description (FD) - 3.6.7 Review Database Specification (DB).
NOTE: Data reverse engineering is not addressed in this investigation.

a. Review FD Sections 2, 3, and 4; extract relevant portions.

b. Review SS Sections 2 and 4; extract relevant portions.

c. Review SSS Sections 2 and 4; extract relevant portions.

d. Review US or PS Sections 2 and 3; extract relevant portions by program identification.

e. Review MM Section 2; extract high-level system structure information.

f. Review OM Sections 2 and 3; extract relevant portions.

g. Review UM Section 4 for potentially useful information; extract if found.

h. Place extracted documentation in the RE library.

i. Update document index information.

j. Update component-document information for documents that pertain specifically to a particular component.

Tables/Identifiers:

Document (13)	Component-Document (7)
No-Seq-Doc (I)	ID-Component (F)
	No-Seq-Doc (F)

4.1.1 Assign Facilitator - 4.1.2 Assign Modeling Specialist.

a. Review qualifications of available reverse engineers.

b. Assign reverse engineer as facilitator to lead the domain analysis modeling group.

c. Assign reverse engineer as a data modeling specialist to support the domain analysis model effort.

Tables/Identifiers: None.

4.1.3 Select Functional Analyst - 4.1.4 Select Technical Analyst.

a. Determine major domain areas represented in the target system.

b. Poll managers for nominations for functional analysts familiar with the domain areas.

c. Poll operations managers for nominations for technical analysts familiar with the target system.

d. Review qualifications of functional and technical analysts.

e. Select functional analyst for each domain area.

f. Select technical analysts familiar with major subsystems of the target system.

g. Coordinate selections with appropriate managers.

Tables/Identifiers: None.

4.1.5 Prepare Modeling Schedule - 4.2.1.4 Establish Domain Activity.

a. Prepare preliminary domain modeling schedule based on anticipated complexity of the application domain and four-hour modeling sessions.

b. Notify participants of modeling session times and locations.

c. Conduct modeling training session to familiarize participants with the concepts of functional decomposition and the procedures used to develop an outline function model.

d. Identify the major key areas of the application domain. Between seven and nine key areas are identified and serve as the major subdivisions of the application domain area. All domain area specialists and functional user analysts selected for the domain modeling exercise are present during modeling sessions. After the key

areas have been established and validated, only those personnel with knowledge in a particular key area participate in the modeling sessions. Identify key areas with single digit numbers.

e. Establish tasks within each key area. Tasks are lower-level functions carried out to complete a key area. Between five and nine tasks are identified for each key area. Tasks are identified with two-digit numbers.

f. Establish subtasks within each task. Subtasks are lower-level functions performed to complete a task. Subtasks are optional and may appear at multiple levels depending on the complexity of the upper level function. Between five and nine subtasks may be established for a task or another subtask. Subtasks have a minimum three-digit identifying number.

g. Establish activities. Activities are the lowest level functions contained in the domain model and represent specific tasks executed to satisfy the task or subtask at the next higher level. Activities normally represent things people do, have specific start and stop points, have clear inputs, and result in clear outputs. Activities may be a higher than normal level in an application domain model. Activities have a minimum three-digit identifying number, but may have more if there are subtasks between tasks and activities.

Tables/Identifiers: None.

4.2.2 Validate Outline Domain Model - 4.2.3 Revise Outline Domain Model.

a. Format draft outline domain model.

b. Distribute draft outline domain model to other functional users for review and comment.

c. Review comments and suggested changes to outline domain model with the domain modeling group.

d. Revise the outline domain model according to approved changes.

Tables/Identifiers: None.

4.2.4 Create Described Domain Model - 4.2.5 Validate Domain Model.

a. Conduct a "how to write functional narrative descriptions" training session for the domain modeling group.

b. Make writing assignments. Narrative descriptions for the domain model are written by domain specialists and functional users with the best qualifications in a particular area. Individuals--not groups--write narrative descriptions.

c. Review narrative descriptions for uniformity and proper level of detail. Return descriptions to authors, as required, for corrections and updates.

d. Consolidate narrative descriptions and format for distribution.

e. Distribute described domain model to organizational personnel for review and comments.

Tables/Identifiers: None.

4.2.6 Prepare Final Domain Model - 4.2.7 Publish Domain Model.

a. Review comments and suggested changes to the domain model with the domain modeling group.

b. Revise the outline domain model according to approved changes.

c. Prepare domain model for publishing.

d. Distribute domain model to functional users and reverse engineers.

e. Store the domain model in the RE library.

Tables/Identifiers:

Process (26)	Process-Process (28)
No-Seq-Process (I)	Is-Parent-Of (F)
	Is-Child-Of (F)

4.3.1 Identify Major Domain Element - 4.3.3 Identify Related Systems.

a. Identify candidate domain element.

b. Consult with functional users and domain specialists.

c. Define domain element.

d. Define relationship with other domain elements using functional user and domain specialist input.

e. Identify interfacing systems.

f. Identify related (but not interfacing) systems as appropriate.

g. Define interfacing and related systems as domain objects.

h. Store object information in the RE library.

Tables/Identifiers:

Object (22)
No-Seq-Object (I)

4.3.4 Prepare Draft Technical Model - 4.3.7 Publish Technical Model.

a. Interview technical personnel with knowledge of the target system technical environment.

b. Model the current technical environment of the target system.

c. Prepare draft technical model (graphic and narrative).

d. Distribute the technical model to technical personnel for review and comment.

e. Review the comments and recommended changes to the model.

f. Prepare final technical model by incorporating recommended changes.

g. Publish the technical model and provide to members of the reverse engineering team.

h. Store the technical model in the RE library.

Tables/Identifiers: None.

4.4.1 Identify Object - 4.5 Establish Project Dictionary.

a. Identify candidate domain object (captures a semantic primitive in the application domain).

b. Consult with functional users to validate the object.

c. With the assistance of a knowledgeable functional user, define the object.

d. Store the object and definition in the RE library.

Tables/Identifiers:

Object (22)
No-Seq-Object (I)

4.5.1 Define Acronym - 4.5.2 Define Term.

a. Record unknown acronym.

b. Record unknown term.

c. Consult with functional personnel to identify and define unknown acronyms and terms.

d. Store defined acronyms and terms in the RE library.

Tables/Identifiers:

Acronym (1)	Term (34)
No-Seq-Acronym (I)	No-Seq-Term (I)

4.5.3 Prepare Acronym Report.

a. Extract acronyms from RE library.

b. Sort in alphabetical sequence.

c. Format acronym report.

d. Distribute to reverse engineers.

Tables/Identifiers:

Acronym (1)
No-Seq-Acronym (I)

4.5.4 Prepare Term Report.

a. Extract terms from RE library.

b. Sort in alphabetical sequence.

c. Format terms report.

d. Distribute to reverse engineers.

Tables/Identifiers:

Term (34)
No-Seq-Term (I)

5.1.1 Group Component - 5.1.2 Assign Reverse Engineer.

a. Using the system technical diagram, assign a group number to each component in the target system.

b. Assign a reverse engineer by name to each component in the target system

Tables/Identifiers:

Component (4)
ID-Component (I)

5.1.3 Print Component Summary Report.

a. Extract component data from the RE library.

b. Format the component summary report for each target system component

c. Distribute printed reports to reverse engineers and the reverse engineering project manager.

Tables/Identifiers:

Component (4)
ID-Component (I)

Component-Comment (5)
ID-Component (F)
No-Seq-Comment (F)

Comment (3)
No Seq-Comment (I)

5.1.4 Record Component Status.

a. Retrieve specific component data from the RE library.

b. Update reverse engineering percent complete.

Tables/Identifiers:

<u>Component</u> (4)
ID-Component (I)

5.2.1 Review Identification Division.

a. Read batch COBOL component identification division.

b. Evaluate introductory comments.

c. Record comments if they explain or clarify the component and have not been previously recorded (type = I).

d. Write explanatory comment if necessary (type = A); multiple comments may be written during the reverse engineering effort.

e. Record the beginning line number of type I comment.

Tables/Identifiers:

<u>Comment</u> (3)
No-Seq-Comment (I)

<u>Component-Comment</u> (5)
ID-Component (F)
No-Seq-Comment (F)

5.2.2 Review Input-Output Section.

a. Read batch COBOL component environment division file section.

b. Search for SELECT statements.

c. Extract internal file name (identifier following SELECT).

d. Make list of internal file names.

e. Extract external file name (identifier following ASSIGN TO). May refer to a device rather than a physical file name (e.g., SYS006-UT-2400-S), in which case the file name should be retrieved from the JCL.

f. Extract file organization; default is sequential.

g. Extract access mode (sequential, random, or dynamic); default is sequential.

h. Determine file type (system internal = I, system external = E).

i. Determine file media.

Tables/Identifiers:

<u>File</u> (15)
ID-File (I)

<u>Component-File</u> (8)
ID-Component (F)
ID-File (F)

5.2.3 Review File Section.

a. Read batch COBOL component data division file section.

b. Search for FD statements for each file in file list.

c. Find 01 level record name for an FD statement.

d. Extract record name (identifier following 01 level designator).

e. Extract layout for file records.

f. Extract data elements for records.

Tables/Identifiers:

<u>File-Record</u> (16)
ID-File (I)

<u>Record</u> (29)
No-Record-Seq (I)
Doc-ID-Code (F)

<u>Record-Element</u> (30)
No-Seq-Record (F)
No-Seq-Element (F)

<u>Element</u> (14)
No-Seq-Element (I)

5.2.4 Review Working Storage Section.

a. Read batch COBOL component data division working storage section.

b. Find DATA-VIEW statement.

c. Record name of variable following DATA-VIEW (e.g., DVCTF02F) as a table used by this component.

d. Find in-line comment.

e. Record in-line comment and its line number (type = I, internal).

f. Find data record identified by a comment, if any.

Tables/Identifiers:

<u>Table</u> (32)
No-Seq-Table (I)

<u>Component-Table</u> (11)
ID-Component (F)
Name-Table (F)

<u>Comment</u> (3)
No-Seq-Comment (I)

<u>Component-Comment</u> (5)
ID-Component F)
No-Seq-Comment (F)

5.2.5.1 Find File OPEN Statement.

a. Read batch COBOL component procedure division.

b. Find OPEN statement.

c. Identify file activity (e.g., input, output, input-output).

d. Record file activity (II, OO, IO).

e. Repeat until all files found in SELECT statements have been matched with an OPEN.

f. Create a File Open Error note for technical personnel (file not matched with an OPEN statement).

Tables/Identifiers:

Component-File (8)
ID-Component (F)
ID-File (F)

5.2.5.2 Find File READ Statement.

a. Read batch COBOL component procedure division.

b. Find READ INTO statement.

c. Identify record format associated with this input file.

d. Repeat for all input files found in an OPEN INPUT statement.

e. Update transaction (if this is a new DIC) and transaction activity (read).

Tables/Identifiers:

File-Record (16)
ID-File (F)
No-Seq-Record (F)

Component-Transaction (12)
ID-Component (F)
Doc-ID-Code (F)

Record (29)
No-Seq-Record (I)
Doc-ID-Code (F)

Transaction (35)
Doc-ID-Code (I)

5.2.5.3 Find File WRITE Statement.

a. Read batch COBOL component procedure division.

b. Find WRITE FROM statement (this statement may not be used in all programs).

c. Identify record format associated with this output file.

d. Update transaction (if this is a new DIC) and activity (create).

e. Repeat for all files found in an OPEN OUTPUT statement.

Tables/Identifiers:

<u>File-Record</u> (16)
ID-File (F)
No-Seq-Record (F)

<u>Record</u> (29)
No-Seq-Record (I)
Doc-ID-Code (F)

<u>Component-Transaction</u> (12)
ID-Component (F)
Doc-ID-Code (F)

<u>Transaction</u> (35)
Doc-ID-Code (I)

5.2.5.4 Find Database Table Name.

a. Read batch COBOL component procedure division.

b. For each table used by the component, find the table name.

c. Determine the use of the table (create, read, update, delete).

d. Record the actual table name and prefix if given in a comment field and not already identified.

Tables/Identifiers:

<u>Component-Table</u> (11)
Component-ID (F)
No-Seq-Table (F)

<u>Table</u> (32)
No-Seq-Table (I)

5.2.5.5.1 Identify Source Paragraph - 5.2.5.5.5 Identify Transform Paragraph.

a. Read batch COBOL component procedure division.

b. Find COBOL paragraph name (e.g., 5250-NSN-CHANGE).

c. Read paragraph note, if present, and paragraph content.

d. Determine paragraph type:

(1) Source (input) - skip.

(2) Sink (output) - skip.

(3) Computation (e.g., COMPUTE PMRU-ACT-QTY = PMRU-CTY * CONV-FAC).

(4) Business rule (e.g. IF condition THEN action).

(5) Transform and all others (may be a combination of computation, business rule, and transforms).

e. Record the paragraph name.

f. Record the paragraph starting line number.

g. Record the paragraph note and type (I = internal).

h. Record the paragraph note starting line number.

i. Write descriptive note, if necessary (type = A, added).

Tables/Identifiers:

<u>Component-Paragraph</u> (9)
ID-Component (F)
No-Seq-Para (F)

<u>Note</u> (21)
No-Seq-Note (I)

<u>Paragraph</u> (3)
No-Seq-Para (I)

<u>Paragraph-Note</u> (25)
No-Seq-Para (F)
No-Seq-Note (F)

5.2.5.6 Find Document Identifier Code (DIC).

a. Read batch COBOL component procedure division.

b. Find three-character document identifier codes (e.g., ZAK, XAA, ZSC) in text (e.g., IF INPUT-DOC = "XAA") or in comments (e.g., * THIS PARAGRAPH PROCESSES THE XAA RECORD).

c. Determine activity with respect to the DIC (create, read, update).

d. Add DIC if it is not already in the list of transactions; if available, also add a description of the transaction.

Tables/Identifiers:

<u>Component-Transaction</u> (12)
ID-Component (F)
Doc-ID-Code (F)

<u>Transaction</u> (35)
Doc-ID-Code (I)

5.2.5.7 Identify Called Component.

a. Read batch COBOL component procedure division.

b. Find CALL statements.

c. Extract component name (in quotes following the CALL statement).

d. Find USING statement, if present.

e. Record the parameters following the USING statement as a character string with each item separated by a comma and space (e.g., "X, Y, Z").

f. Set data pass to Y if a USING statement was found.

g. Set data pass to N if a USING statement was not found.

Tables/Identifiers:

<u>Component-Component</u> (6)
ID-Component-Calls (F)
ID-Component-Called (F)

5.3.1 Review CICS Identification Division.

a. Read CICS COBOL component identification division.

b. Evaluate introductory comments.

c. Record comments if they explain or clarify the component and have not been previously recorded (type = I).

d. Write explanatory comment if necessary (type = A).

e. Record the beginning line number of type I comments.

Tables/Identifiers:

<u>Comment</u> (3)
No-Seq-Comment (I)

<u>Component-Comment</u> (5)
ID-Component (F)
No-Seq-Comment (F)

5.3.2 Review CICS Working Storage Section.

a. Read CICS COBOL component data division working storage section.

b. Find DATA-VIEW statement.

c. Record name of variable following DATA-VIEW (e.g., DVCTF02F) as a table used by this component.

d. Find in-line comment.

e. Record in-line comment and its line number (type = I, internal).

f. Find data record identified by a comment, if any.

Tables/Identifiers:

<u>Table</u> (32)
No-Seq-Table (I)

<u>Component-Table</u> (11)
ID-Component (F)
Name-Table (F)

<u>Comment</u> (3)
No-Seq-Comment (I)

<u>Component-Comment</u> (5)
ID-Component F)
No-Seq-Comment (F)

5.3.3.1 Find CICS File Read Statement.

a. Read batch COBOL component procedure division.

b. Find CICS read statement for virtual storage access method (VSAM) file.

Format is EXEC CICS XXXXXXXXX where XXXXXXXXX may be:

READ		READPREV	(read previous)
STARTBR	(start browse)	RESETBR	(reset browse)
ENDBR	(end browse)	READUPDATE	(read for update)
READNEXT	(read next record)		

c. Identify the file name associated with this read command. Following the read command is a DATASET (ddname) statement. The name of the file is "ddname."

d. Identify record format associated with this input file. Following the DATASET(ddname) statement is an INTO (area-name) statement. The "area-name" represents the record for this input file.

e. Update transaction (if this is a new DIC) and transaction activity (read).

Tables/Identifiers:

<u>File-Record</u> (16)
- ID-File (F)
- No-Seq-Record (F)

<u>Record</u> (29)
- No-Seq-Record (I)
- Doc-ID-Code (F)

<u>Component-Transaction</u> (12)
- ID-Component (F)
- Doc-ID-Code (F)

<u>Transaction</u> (35)
- Doc-ID-Code (I)
- Description-DIC

<u>File</u> (15)
- ID-File (I)

<u>Component-File</u> (8)
- ID-Component (F)
- ID-File (F)

5.3.3.2 Find CICS File Write Statement.

a. Read CICS COBOL component procedure division.

b. Find CICS write statement for virtual storage access method (VSAM) file.

Format is EXEC CICS XXXXXXXX where XXXXXXXX may be:

WRITE
REWRITE
UNLOCK

c. Extract the file name. Following the write command is a DATASET (ddname) statement. The file name is "ddname."

d. Identify record format associated with this output file. Following the DATASET (ddname) statement is a FROM (area-name) statement. The record for this output file is "area-name."

e. Update transaction (if this is a new DIC) and activity (create).

Tables/Identifiers:

File-Record (16)
- ID-File (F)
- No-Seq-Record (F)

Record (29)
- No-Seq-Record (I)
- Doc-ID-Code (F)

Component-Transaction (12)
- ID-Component (F)
- Doc-ID-Code (F)

Transaction (35)
- Doc-ID-Code (I)

File (15)
- ID-File (I)

Component-File (8)
- ID-Component (F)
- ID-File (F)

5.3.3.3 Find CICS File Delete Statement.

a. Read CICS COBOL component procedure division.

b. Find EXEC CICS DELETE statement for VSAM file.

c. Extract the file name associated with this input-output file. Following the delete command is a DATASET (ddname) statement. The name of the file is "ddname." The record associated with the file is identified by other file commands.

Tables/Identifiers:

Component-Transaction (12) ID-Component (F) Doc-ID-Code (F)	Transaction (35) Doc-ID-Code (I)
File (15) ID-File (I)	Component-File (8) ID-Component (F) ID-File (F)

5.3.3.4 Find CICS Database Table.

a. Read CICS COBOL component procedure division.

b. For each table used by the component, find the table name.

c. Determine the use of the table (create, read, update, delete).

d. Record the actual table name and prefix if given in a comment field and not already identified.

Tables/Identifiers:

Component-Table (11) Component-ID (F) No-Seq-Table (F)	Table (32) No-Seq-Table (I)

5.3.3.5 Find Terminal Statement.

a. Read CICS COBOL procedure division.

b. Find CICS terminal statements. Terminal statements take the form EXEC CICS XXXXXXXXX, where XXXXXXXXX is one of the following:

RECEIVE MAP (map-name) MAPSET (mapset-name) INTO (data-area)

SEND MAP (map-name) MAPSET (mapset-name) FROM (data-area)
RECEIVE INTO (data-area)
SEND FROM (data-area)

c. Identify screen. Use the "map-name" value as the screen name.

d. Set screen type to C (CICS).

e. Identify screen activity (input or output) from the SEND or RECEIVE command or display only from the BROWSE command.

Tables/Identifiers:

Component-Screen (10)	Screen (31)
ID-Component (F)	ID-Screen (I)
ID-Screen(F)	

5.3.4.1 Identify CICS Source Paragraph - 5.3.4.5 Identify CICS Transform Paragraph.

a. Read CICS COBOL component procedure division.

b. Find COBOL paragraph name (e.g., 0200-PROCESS-TRANS).

c. Read paragraph note, if present, and paragraph content.

d. Determine paragraph type:

(1) Source (input) - skip.

(2) Sink (output) - skip.

(3) Computation (e.g., COMPUTE ACTUAL-QTY = EST-QTY * CONV-FAC).

(4) Business rule (e.g. IF NUMBER-ENTRIES > 2 THEN action).

(5) Transform and all others (may be a combination of computation, business rule, and transforms).

e. Record the paragraph name.

f. Record the paragraph starting line number.

g. Record the paragraph note and type (I = internal).

h. Record the paragraph note starting line number.

i. Write descriptive note, if necessary (type = A, added).

Tables/Identifiers:

Component-Paragraph (9) ID-Component (F) No-Seq-Para (F)	Note (21) No-Seq-Note (I)
Paragraph (3) No-Seq-Para (I)	Paragraph-Note (25) No-Seq-Para (F) No-Seq-Note (F)

5.3.4.6 Find CICS Document Identifier Code (DIC).

a. Read CICS COBOL component procedure division.

b. Find three-character document identifier codes (e.g., A01, A02) in text (e.g., IF INPUT-DOC = "XAA") or in comments (e.g., * CHECK DATE ON A01).

c. Determine activity with respect to the DIC (create, read, update).

d. Add DIC if it is not already in the list of transactions; if available, add a description of the transaction.

Tables/Identifiers:

Component-Transaction (12) ID-Component (F) Doc-ID-Code (F)	Transaction (35) Doc-ID-Code (I)

5.3.4.7 Identify CICS Called Component.

a. Read CICS COBOL component procedure division.

b. Find EXEC CICS LINK PROGRAM ('module-name') statement (implies return to calling program) or EXEC CICS XCTL ('module-name') statement (does not return control to calling program).

c. Extract component name in quotes following LINK PROGRAM statement.

d. Find COMMAREA statement following LINK PROGRAM ('module-name') statement, if present. If not present, no data is passed to the called component.

e. Leave the parameters for the call blank (detailed analysis of program logic is required to determine values established in the COMMAREA before the call is made).

f. Set data pass to Y if a COMMAREA statement was found.

g. Set data pass to N if a COMMAREA statement was not found.

Tables/Identifiers:

Component-Component (6)
ID-Component-Calls (F)
ID-Component-Called (F)

5.4.1 Review Header.

a. Read 4GL component header (identified by ->PROGRAM module-name).

b. Find comments in SHORT-DESC 'text' statements, TEXT *n* statements or following : (colon).

c. Record comments if they explain or clarify the component and have not been previously recorded (type = I).

d. Write explanatory comment if necessary (type = A).

e. Record the beginning line number of type I comment (reference listings of IDEAL programs have line numbers added).

f. Find USES-DATAVIEW statement. Record the database table name that follows the USES statement (e.g., DVCTF02F).

g. Find USES-PROGRAM statement.

h. The module name following the USES-PROGRAM statement is a called program for this component.

Tables/Identifiers:

Comment (3)
No-Seq-Comment (I)

Component-Comment (5)
ID-Component (F)
No-Seq-Comment (F)

Table (32)
No-Seq-Table (I)

Component-Table (11)
ID-Component (F)
Name-Table (F)

Component-Component (6)
ID-Component-Calls (F)
ID-Component-Called (F)

5.4.2 Review Working Data.

a. Read 4GL working data section (identified by ->WORKING DATA).

b. Find in-line note (identified by : (colon)).

c. Record in-line comment and line number.

d. Set comment type to internal.

Tables/Identifiers:

Comment (3)
No-Seq-Comment (I)

Component-Comment (5)
ID-Component F)
No-Seq-Comment (F)

5.4.3 Review Parameter Data.

a. Read 4GL parameter data section (identified by ->PARAMETER DATA).

b. Find in-line comment (identified by : (colon) characters).

c. Record in-line comment and its line number (type = I, internal).

Tables/Identifiers:

Comment (3)	Component-Comment (5)
No-Seq-Comment (I)	ID-Component F)
	No-Seq-Comment (F)

5.4.4.1 Review Main Procedure Data.

a. Read 4GL procedure data section (identified by ->PROCEDURE DATA and

<<MAIN>> PROCEDURE statements).

b. Find in-line comment (identified by one or more : (colon) characters).

c. Record in-line comment and its line number (type = I, internal).

Tables/Identifiers:

Comment (3)	Component-Comment (5)
No-Seq-Comment (I)	ID-Component F)
	No-Seq-Comment (F)

5.4.4.2 Find 4GL Database Table.

a. Read 4GL component procedure data (identified by ->PROCEDURE DATA).

b. Find the table name for each table used by the component.

c. Determine the use of the table (create, read, update, delete).

d. Record the actual table name and prefix if given in a comment field and not

already identified.

Tables/Identifiers:

Component-Table (11)	Table (32)
Component-ID (F)	No-Seq-Table (I)
No-Seq-Table (F)	

5.4.4.3 Find 4GL Terminal Statement.

a. Read 4GL component header (identified by ->PROGRAM program-name).

b. Find USES-PANEL statement. Terminal screens in IDEAL are called panels.

c. The component name following the USES-PANEL statement is the name of an IDEAL program that generates a screen of the same name (e.g., USES-PANEL ZZNAI102).

d. Record the screen name in the RE library and set the type to I (IDEAL).

e. Read 4GL procedure data (identified by <<MAIN>>PROCEDURE and by individually identified procedures).

f. Find TRANSMIT, REFRESH, and SET statements for each screen identified in a USES-PANEL statement to verify screen use (e.g., TRANSMIT ZZNAI102, REFRESH ZZNAI102, SET ZZNAI102 = DVT220U BY NAME). Set activity to DS = display or IO = input/output.

Tables/Identifiers:

Component-Screen (10)	Screen (31)
ID-Component (F)	ID-Screen (I)
ID-Screen (F)	

5.4.4.4.1 Identify 4GL Source Procedure - 5.4.4.4.5 Identify 4GL Transform Procedure.

a. Read 4GL component procedure data (identified by ->PROCEDURE DATA).

b. Find 4GL procedure name. Procedures begin with a name and the word PROCEDURE (e.g., <<P700-INITIALIZE>> PROCEDURE) and end with ENDPROC. IDEAL procedures are executed with a DO statement, much like the COBOL PERFORM.

c. Read procedure note, if present, and procedure content.

d. Determine paragraph type:

(1) Source (input) - skip.

(2) Sink (output) - skip.

(3) Computation (e.g., COMPUTE QTY = PMRU-ACTY * FAC)

(4) Business rule (e.g., IF condition THEN action).

(5) Transform and all others.

e. Record procedure name.

f. Record procedure starting line number.

g. Record procedure note and type (I = internal).

h. Record procedure note starting line number.

i. Write descriptive note, if necessary (type = A, added).

Tables/Identifiers:

<u>Component-Paragraph</u> (9)
ID-Component (F)
No-Seq-Para (F)

<u>Note</u> (21)
No-Seq-Note (I)

<u>Paragraph</u> (3)
No-Seq-Para (I)

<u>Paragraph-Note</u> (25)
No-Seq-Para (F)
No-Seq-Note (F)

5.4.4.4.6 Find 4GL Document Identifier Code (DIC).

a. Read 4GL component procedure data (identified by ->PROCEDURE DATA).

b. Find three-character document identifier codes (e.g., ZAK, XAA, ZSC) in text (e.g., IF DOC = "XAA") or in comments (e.g., * PROCESSES THE XAA).

c. Determine activity with respect to DIC (create, read, update).

d. Add this DIC if it is not already in the list of transactions; if available, add a description of the transaction.

Tables/Identifiers:

<u>Component-Transaction</u> (12)
ID-Component (F)
Doc-ID-Code (F)

<u>Transaction</u> (35)
Doc-ID-Code (I)

4.4.4.7 Identify 4GL Called Component.

a. Read 4GL component procedure data (identified by ->PROCEDURE DATA).

b. Find CALL statements.

c. Extract component name following CALL statement (e.g., CALL ZZOT0123).

d. Find USING statement, if present.

e. Record parameters following USING statement as a character string with each item separated by a comma and space (e.g., "X, Y, Z").

f. Set data pass to Y if a USING statement was found.

g. Set data pass to N if a USING statement was not found.

Tables/Identifiers:

<u>Component-Component</u> (6)
ID-Component-Calls (F)
ID-Component-Called (F)

6.1.1 Verify Component Status - 6.1.2 Print Program Model.

a. Verify the initial component review has been completed. If not complete, do not proceed; complete initial review.

b. Prepare the program model listing (see Figure 27).

Tables/Identifiers:

<u>Transaction</u> (35)
Doc-ID-Code (I)

<u>Table</u> (32)
No-Seq-Table (I)

<u>Comment</u> (3)
No-Seq-Comment (I)

<u>Component</u> (4)
ID-Component (I)

<u>Component-Comment</u> (5)
ID-Component (F)
No-Seq-Comment (F)

<u>Component-Component</u> (6)
ID-Component-Calls (F)
ID-Component-Called (F)

<u>Component-File</u> (8)
ID-Component (F)
ID-File (F)

<u>Component-Paragraph</u> (9)
ID-Component (F)
No-Seq-Para (F)

<u>Component-Screen</u> (10)
ID-Component (F)
ID-Screen (F)

<u>Component-Table</u> (11)
ID-Component (F)
Name-Table (F)

<u>Component-Transaction</u> (12)
ID-Component (F)
Doc-ID-Code (F)

<u>File</u> (15)
ID-File (I)

<u>Note</u> (21)
No-Seq-Note (I)

<u>Paragraph</u> (23)
No-Seq-Para (I)

<u>Paragraph-Note</u> (25)
No-Seq-Para (F)
No-Seq-Note (F)

<u>Screen</u> (31)
ID-Screen (I)

Program-ID:	ZZLAW070.		
Program Name:	Extract backorder Master Data for D016, D032 and Q072.		
Called by:	None.		
Calls:	ZZLAR171	Passes/returns Error codes/error messages.	
Comments (Internal):	I-339. BACKORDER FILE FOR INTERFACE WITH D032. ALSO USED TO GENERATE A SIMILAR FILE FOR INTERFACE WITH D016 AND Q072.		
Comments (Added):	A-000. Program also appears to be modifying estimated shipping dates.		
Input Files:	BATCH-PARM-FILE (SYSIN). Contains control information for generating report for 15th of month or end of month.		
Output Files:	BACKORDER-OUT-FILE (ZZ0070A0) (External). Contains all open backorder records meeting selection criteria established by interfacing systems.		
Database Tables:	DVBOF02U (SCD-BACKORDER-REC)	(R)	Backorders.
	DVITF13R (SCD-ITEMS-REC)	(R)	Item data.
	DVRQF03R (SCD-REQUIS-REC)	(U)	Requisition.
Transactions:	AE3	(C)	ABC transaction.
	AE4	(C)	DEF transaction.
	AE5	(C)	GHI transaction.

Procedure Division Extracts:

01607-I	PROCESS SCD-BACKORDER-REC, CREATE BACKORDER-OUT RECORD.
01610	1100-PROCESS-BO.

Figure 27. Sample program implementation model.

01696-I	CHECK RECORD TO SEE IF IT CONTAINS CRYPTO DATA.
01696-A	Temporary MMC = CA, CI, CS, or XU.
01698	1250-CRYPTO.
01856-I	CHECK TO SEE IF THE STOCK NUMBER HAS CHANGED.
01858	1700-CHECK-STKXREF.
02121-I	MOVE DATA TO BW-WRK BUFFER. LOAD CONSTANT VALUES.
02121-A	This is the main processing routine.
02124	3100-BILD-BO-WRK.
02341-I	DETERMINE THE CORRECT ESTIMATED SHIPPING DATE (ESD).
02243	3125-CHECK-ESD.
02263-I	THE BACKORDER HAS NOT EXPIRED PAST ORIGINAL ESD.
02267-I	THE BACKORDER HAS NOT EXPIRED PAST CALCULATED ESD.
02272-I	THE BACKORDER HAS EXPIRED PAST CALCULATED ESD.
02305-I	COMPUTES THE CORRECT ESD FOR THE BACKORDER.
02307	3150-COMPUTE-ESD.
02651	5100-PROCESS-AE3-TRANS.
02661-I	THIS PROCEDURE CHECKS THE PROCESS SWITCHES AND CALLS THE APPROPRIATE PROCEDURE TO CONTINUE PROCESSING.
02664	5500-COMPL-TRANS-PROC.
02740-I	THIS PROCEDURE PROCESSES SAP-AE TRANSACTIONS.
02742	5520-PROCESS-SAP-AE.

Figure 27. (continued)

6.1.3 Retrieve Program Reference Listing.

a. Retrieve the printed program reference listing from the RE library.

b. Check listing to ensure it is complete. If the listing is for an IDEAL component, ensure it has been printed with continuous line numbers.

Tables/Identifiers: None.

6.1.4 Print Documentation List - 6.1.5 Print Contact Point.

a. Print the list of available documentation.

b. Print assistance contact points.

Tables/Identifiers:

Component (4)
ID-Component (I)

Component-Document (7)
ID-Component (F)
No-Seq-Doc (F)

Document (13)
No-Seq-Doc (I)

Comments

6.2.1 Review Program Model.

a. Review the initial program model.

b. Ensure critical information has been included.

c. Note discrepancies.

Tables/Identifiers: None.

6.2.2 Review Documentation.

a. Using the list of documentation available for the component, review available documents and document extracts.

b. Identify component purpose.

c. Identify component objectives.

d. Identify assumptions and constraints.

e. Update the RE library.

Tables/Identifiers:

Component (4)
ID-Component (I)

6.2.3 Review Source Code.

a. Review source code for familiarization using the printed reference listing.

b. Compare the source listing with the program model. Randomly check paragraph and note line numbers from the program model with the source code listing.

c. Adjust the program model, if necessary, and reprint it.

Tables/Identifiers: None.

6.2.4 Prepare Input-Output Diagram - 6.2.9 Produce Final Implementation Model.

a. Prepare a data flow context diagram for the component; show inputs and outputs, tables, and interface files.

b. Validate input and output shown in the program model with source code.

c. If necessary, consult with technical and domain specialists to resolve discrepancies or to enhance understanding of the component.

d. Resolve discrepancies between documentation, source code, and the initial implementation model.

e. Identify changes required.

f. Modify the RE library as necessary to correct the implementation model.

g. Print the final implementation model (see services under 6.1.2).

Tables/Identifiers: Any table in the database.

6.3.1 Segment Component.

a. Review the program implementation model.

b. Group paragraphs in logical groups if the model structure is not already in this form.

c. Record the paragraph group assignments in the RE repository.

d. Ensure that links to subprograms are represented in the logical paragraph structure.

e. Reprint the program implementation model.

Tables/Identifiers:

<u>Component</u> (4)
ID-Component (I)

<u>Component-Paragraph</u> (9)
ID-Component (F)
No-Seq-Para (F)

<u>Paragraph</u> (23)
No-Seq-Para (I)

6.3.2 Identify Key Data Item.

a. Using the program implementation model listing as a guide, review the reference listing to identify the major data structure or structures manipulated in an extracted paragraph.

b. Identify the domain object represented by the data structure.

c. Note the domain object on the program implementation model.

Tables/Identifiers: None.

6.3.3 Create Structural Model.

a. Using the program implementation model, the input-output diagram, and the reference listing, prepare a high-level structural model of the program under review.

b. Show major processing blocks and domain objects represented, as well as the relationships between the processing blocks.

Tables/Identifiers: None.

6.4.1 Analyze Paragraph - 6.4.3 Assign Meaning.

a. Using the program implementation model, structural model, input-output diagram, documentation and input from technical and domain specialists, analyze individual paragraphs in the reference listing.

b. Paragraphs not in the implementation model are reviewed, if necessary, to facilitate understanding of the transform paragraphs.

c. Interpret a transform paragraph and assign functional meaning to it by writing a short paragraph describing the function performed by the code. The description is written in non-technical, domain-oriented terms. Jargon, abbreviations, acronyms, and unique terms are avoided. A properly written functional statement should be no more than a paragraph of three or four sentences.

d. Record the functional narrative in the RE library, associating it with the paragraph in the implementation model it represents.

Tables/Identifiers:

<u>Function</u> (17)
No-Seq-Function

<u>Paragraph</u> (23)
No-Seq-Para (I)

<u>Paragraph-Function</u> (24)
No-Seq-Para (F)
No-Seq-Function (F)

6.5.1 Print Outline Domain Model.

a. The outline domain model prepared at the beginning of the reverse engineering analysis is extracted from the RE library.

b. Print one copy of the domain model in normal spacing; this version is used as a guide for allocating functions to the model.

c. Print one copy of the domain model with each activity (primitive-level function) placed on a separate page. This version is used to allocate extracted functions to a specific area in the domain model.

Tables/Identifiers:

<u>Process</u> (26)
No-Seq-Process (I)

<u>Process-Process</u> (28)
Is-Parent-Of (F)
Is-Child-Of (F)

6.5.2 Produce Draft Function Model.

a. Verify all components in the target system have been analyzed (percent complete is equal to 100).

b. Print a list of the functions extracted from each system component. A separate function list is prepared for each component.

c. Assign a function to the domain model by writing the unique function number under the appropriate activity number in the printed domain model.

d. When all extracted functions have been assigned to an activity, review the assignments to ensure there are no more than nine subelements under any activity.

e. Introduce new subtasks into the domain model, if necessary, to preserve the seven-plus or minus two rule of hierarchical structure.

f. Reassign extracted function (numbers) to newly created subtasks.

g. Update the domain model structure with added subtasks.

h. Enter the function numbers and associated domain model processes into the RE library. An extracted function is copied to its parent and assigned the parent's number plus another digit (e.g., extracted function assigned to activity 1.2.3 becomes activity 1.2.3.1; activity .1.2.3 becomes subtask 1.2.3 because it now has children).

Tables/Identifiers:

<ins>Component</ins> (4)
ID-Component (I)

<ins>Component-Paragraph</ins> (9)
ID-Component (F)
No-Seq-Para (F)

<ins>Function</ins> (17)
No-Seq-Function (I)

<ins>Paragraph-Function</ins> (24)
No-Seq-Para (F)
No-Seq-Function (F)

<ins>Narrative</ins> (19)
No-Seq-Narrative (I)

<ins>Narrative-Function</ins> (19)
No-Seq-Narrative (F)
No-Seq-Function (F)

6.6.1 Distribute Draft Function Model - 6.6.5 Produce Function Model.

a. Print the function model from the RE library. The function model consists of the hierarchical outline structure from the domain model with the functional narrative extracted from system components.

b. Distribute the draft function model to domain specialists and functional users throughout the organization, soliciting comments, recommendations, and proposed changes. Comments and changes are submitted as individual documents, one entry per document. Establish a suspense date for submitting comments.

c. Review comments as they are received from reviewers. Reject proposed changes that do not reflect specific corrective actions.

d. Organize acceptable comments according to model sections.

e. Consolidate duplicate comments.

f. Reassemble the original group who prepared the domain model.

g. In facilitated modeling sessions, proposed changes to the function model are individually reviewed, discussed, and accepted or rejected for incorporation into the model. Changes not accepted are annotated as to reason and returned to originator. Accepted changes are marked for implementation after the modeling session is completed.

h. Implement approved changes to the function model by revising appropriate entries in the RE library.

i. Print and distribute the final function model to appropriate organization managers and functional users.

Tables/Identifiers:

<u>Function</u> (17)
No-Seq-Function (I)

<u>Paragraph-Function</u> (24)
No-Seq-Para (F)
No-Seq-Function (F)

<u>Narrative</u> (19)
No-Seq-Narrative (I)

<u>Narrative-Function</u> (19)
No-Seq-Narrative (F)
No-Seq-Function (F)

<u>Process</u> (26)
No-Seq-Process (I)

<u>Process-Function</u> (27)
No-Seq-Process (F)
No-Seq-Function (F)

<u>Process-Process</u> (28)
Is-Parent-Of (F)
Is-Child-Of (F)

6.7.1 Prepare Context Diagram - 6.7.3 Describe Key Area.

a. Using the narrative from the function model as a starting point, prepare a data flow context diagram for the proposed new system. A context diagram represents the entire system as a single process, identifies the major sources and destinations of data and the major data flows entering and leaving the system.

b. Prepare a level 0 diagram by showing how the key areas from the function model are related to each other and how each interfaces with the environment through data sources and destinations.

Tables/Identifiers: None.

7.1.1.1 - 7.1.2.6 Data modeling activities. Omitted.

7.2.1 Format Major System Components.

a. Assemble the proposed new system context diagram, the level 0 data flow diagram, and the narrative description of the function model key areas.

b. Format for inclusion in paragraph 5.3.d of the Operational Concept Document.

Tables/Identifiers: None.

7.2.2 Describe External Interface.

a. Extract information from the RE library for external files (type E).

b. Combine repository information with interface agreement details.

c. Format material for inclusion in paragraph 5.3.c of the Operational Concept Document.

Tables-Attributes:

<u>File</u> (15)
ID-File (I)

7.2.3 Format System Function.

a. Print functional key areas, tasks, and subtasks (select process KA, TA, ST).

b. Format for inclusion in paragraph 5.3.d of the Operational Concept Document.

Tables/Identifiers:

<u>Function</u> (17)
No-Seq-Function (I)

<u>Process</u> (26)
No-Seq-Process (I)

<u>Process-Process</u> (28)
Is-Parent-Of (F)
Is-Child-Of (F)

<u>Paragraph-Function</u> (24)
No-Seq-Para (F)
No-Seq-Function (F)

<u>Process-Function</u> (27)
No-Seq-Process (F)
No-Seq-Function (F)

7.2.4 Format Functional Hierarchy.

a. Print function model key areas, tasks, subtasks and activities from the RE repository (select process KA, TA, ST, and AC).

b. Combine function model and data flow diagrams.

c. Format for inclusion in paragraph 5.3.e of the Operational Concept Document.

Tables/Identifiers:

Function (17)
- No-Seq-Function (I)

Paragraph-Function (24)
- No-Seq-Para (F)
- No-Seq-Function (F)

Narrative (19)
- No-Seq-Narrative (I)

Narrative-Function (19)
- No-Seq-Narrative (F)
- No-Seq-Function (F)

Process (26)
- No-Seq-Process (I)

Process-Function (27)
- No-Seq-Process (F)
- No-Seq-Function (F)

Process-Process (28)
- Is-Parent-Of (F)
- Is-Child-Of (F)

7.3.1-7.3.2 Data modeling activities. Omitted.

The Logical Data Model

The logical data model was created by populating skeletal tables (developed during the definitional model phase) with attributes. Attributes were identified during logical process modeling.

Populated tables for the reverse engineering methodology support tool are described in Appendix C. These tables are all in third normal form.

The logical data model was represented schematically by a table diagram showing paths between tables (see Figure 28). The paths are links between table identifiers. The diagram shows table names, table numbers, and primary and foreign keys. Keys are located at the top of the table block. Associative entity tables (join tables) have two foreign keys making up the identifier. Two entity tables (Component (4) and Record (29)) have foreign keys. Identifiers in these two tables are designated with (I); the foreign keys are identified by (F). Directed lines on the diagram show links between tables based on identifiers; the arrowheads point to foreign keys.

The logical data model can be implemented in any relational database management system with few changes. The conceptual and logical data models along with the detailed table descriptions, for example, contain sufficient detail to implement the RE library using one of several personal computer-based database management systems (e.g., Access, Paradox, Approach, FoxPro).

ID-Screen
Screen 31

ID-Screen
ID-Component
Component-Screen 10

Doc-ID-Code
Transaction 35

Doc-ID-Code
ID-Component
Component-Transaction 12

No-Seq-Doc
ID-Component
Component-Document 7

No-Seq-Doc
Document 13

No-Seq-Para
No-Seq-Function
Paragraph-Function 24

No-Seq-Table
Table 32

ID-Component
No-Seq-Table
Component-Table 11

No-Seq-Table
No-Seq-Attribute
Table-Attribute 33

No-Seq-Para
ID-Component
Component-Paragraph 9

No-Seq-Para
Paragraph 23

No-Seq-Para
No-Seq-Note
Paragraph-Note 25

ID-Component (I)
No-Seq-Metric (F)
Component 4

ID-Component
No-Seq-Comment
Component-Comment 5

No-Seq-Comment
Comment 3

No-Seq-Note
Note 21

No-Seq-Attribute
Attribute 2

ID-Component
ID-File
Component-File 8

ID-Component-Calls
ID-Component-Called
Component-Component 6

No-Seq-Function
No-Seq-Narrative
Narrative-Function 20

No-Seq-Narrative
Narrative 19

No-Seq-Function
Function 17

No-Seq-Object
Object 22

No-Seq-Metric
Metric 18

No-Seq-Record
No-Seq-Element
Record-Element 30

No-Seq-Element
Element 14

No-Seq-Acronym
Acronym 1

ID-File
File 15

ID-File
No-Seq-Record
File-Record 16

No-Seq-Record (I)
Doc-ID-Code (F)
Record 29

No-Seq-Function
No-Seq-Process
Process-Function 27

No-Seq-Term
Term 34

Is-Parent-Of
Is-Child-Of
Process-Process 28

No-Seq-Process
Process 26

Figure 28. Logical data model.

Formats for Presenting Results

The reverse engineering methodology developed in this chapter is presented as a structured hierarchy of techniques and procedures to be followed in extracting design information from source code. The hierarchy allows the methodology to be viewed at multiple levels. A methodology visual process model augments the narrative descriptions. Conceptual and logical data models portray data structure required to support reverse engineering processes.

Chapter IV describes the application of the the reverse engineering methodology to a small subcomponent of an actual military logistics system. The results (the as-built model) were presented in a conceptual process model representing the functional design information extracted from the case study programs. The as-built model was used to assess the reverse engineering methodology.

Metrics to support reverse engineering time estimates are also presented in Chapter IV. These metrics are presented in the form of effort (time) per line of code by various program types and sizes and are based on data collected during methodology application.

Projected Outcomes

The reverse engineering methodology was defined in detail in this chapter, but was not tested against actual programs. The following outcomes were anticipated following methodology application to the case study.

1. New methodology activities will be identified as a result of finding additional sources of design information in the program code.
2. The effectiveness of a manual process over a computer-based process will be demonstrated.
3. Major functional components will be extracted from the source code.
4. Internal and external interfaces will be identified from the source code.

Resource Requirements

Minimal resource requirements were needed to apply the reverse engineering methodology. Source code from the system selected for analysis was extracted from the mainframe system and loaded as ASCII text on the disk drive of a personal computer. Programs and JCL were printed for review. Microsoft Word for Windows was used to augment the printed programs and JCL with a search function on electronic versions of the subject programs. No other resources were required to support the investigation.

Reliability and Validity

Reliability of the reverse engineering methodology was difficult to measure. The main driver of reliability is the reverse engineer, not the methodology itself. Reliability depends on the reverse engineer's training, experience, domain knowledge, and intuition.

Validity of the methodology also depends on the skill of the reverse engineer, but is easier to objectively measure. Validity was established by comparing extracted design information with known design information.

Summary

This chapter explained the methodology used to develop a practical reverse engineering methodology. Forward engineering and reverse engineering models were compared to substantiate that reverse engineering is not the logical reverse of forward engineering.

Reverse engineering research techniques, methodologies, and tools were reviewed and analyzed. Five specific techniques were examined for possible use in a practical reverse engineering methodology. Although none of the tools were satisfactory, each contained some positive features.

A reverse engineering methodology was developed by applying the information engineering approach to information systems design. The methodology was presented in the form of narrative descriptions of tasks to be performed to recover design information from legacy systems. The narrative description was augmented by a visual process model of activity interrelationships and by conceptual and logical data models required to accomplish reverse engineering. The plan for evaluating the methodology by applying it to a case study was presented and explained.

Chapter IV

Results

This chapter addresses Phase 3 (Case Study Subject Selection), Phase 4 (Reverse Engineering Methodology Application), and Phase 5 (Methodology Assessment). The Data Analysis, Findings, and Summary of Results sections describe the application of the proposed process reverse engineering methodology to a case study of actual COBOL and IDEAL programs.

The Data Analysis section describes the program information database implementation, case study component selection, and methodology application. The Findings section includes methodology assessment, metrics, and proposed methodology changes. The Summary of Results explains the results obtained from the process reverse engineering methodology.

Data Analysis

The reverse engineering methodology developed in Chapter 3 was manually applied to a test case from an operational environment. The objective was to demonstrate methodology feasibility and to evaluate its potential usefulness and applicability to large-scale reverse engineering efforts. The material used in the test case was too voluminous

to be included herein; however, each product specified in the methodology was retained with individual case study components. The sequence of activities was altered slightly to compensate for the lack of knowledgeable functional users to participate in the domain model development. Methodology output products were often incomplete because functional users were unavailable for review, correction, or explanation of unidentified acronyms, document identifier codes, and special terms. Unknown or unclear information is indicated with a question mark (?) in the program information data base and in printed output products.

Implementing the Program Information Database

The program information database described by the logical model in Chapter 3 was implemented in the database management system Microsoft Access Version 2.0 in a Windows 3.1 environment. Using a simple interactive process, Access supported physical data structure development from the logical model. Access' ability to enforce referential integrity was also a significant factor in its selection.

Individual entity and relationship (join) tables were created according to the logical model. Table attribute names were assigned according to the logical model: hyphens were omitted and both upper and lower case letters were used.

After tables were constructed and validated against the model, table associations were established with the Access relationship editor. Relationship details (e.g., referential

integrity rules, cascade deletes, and cascade updates) were also established with the relationship editor.

Using Access Wizards, data entry forms were created for each table in the database for which data was collected. The Component table, the largest in the database, was supported by multiple input forms, each of which contained relevant attributes for a particular activity (e.g., initial program review and program structure analysis).

Thirty-five queries were developed to display data from the database. Acronym lists, database table lists, and document identifier code lists were used frequently during program analysis to support program understanding and to provide a vehicle for collecting additional data.

At the end of the methodology application phase, 3,134 rows had been added to the database. Nine tables (Attribute, Component-Document, Component-Screen, Document, Narrative, Narrative-Function, Object, Screen, and Table-Attribute) were not used because either the data was not available, data modeling was not emphasized, or the table was not required. Two tables (Narrative and Narrative-function) were determined to be redundant.

Selecting Case Study Components

The first step in applying the reverse engineering methodology was to establish selection criteria for a suitable test case. The Stock Control and Distribution system is composed of

nine subsystems, one of which--the Item Manager Wholesale Requisition Process (IMWRP, D035A)--was selected as the test case. The case study was selected using the following criteria:

1. Small to limit the amount of time required to show the methodology's practicality and usefulness.
2. Large enough to gather preliminary metrics to estimate methodology feasibility on large-scale systems.
3. Large enough to uncover weaknesses in the methodology and to suggest enhancements.
4. Representative of system programs and in a related group or subsystem.
5. A large number of interfaces.
6. The same mix of programs as contained in the overall system.
7. Simple enough to permit development of a domain model without functional user assistance.

Nearly 33 million characters of D035A source code were downloaded from the SC&D mainframe system in .TXT format and established in a directory on a personal computer. The source code included COBOL and IDEAL programs, screen formats, and JCL.

Using a high-level list of activities from the D035A functional description, a single component of the system was selected for detailed examination. The Cataloging Management Data (CMD) component is a relatively independent segment of the system

and was easily isolated from other components. CMD contains both batch and on-line programs and is representative of logistics legacy systems.

The system functional description identified 26 major functions associated with catalog management data. A program-to-function allocation list indicated these functions were implemented in 203 program units. However, a review of the program units revealed only 59 unique program identifiers; programs implemented many different functions in a one-to-many relationship. For example, one of the 26 functions was implemented in 19 different program components. The one-to-many relationship between programs and functions suggested extensive reengineering rather than system redesign during the modernization (i.e., existing or restructured programs were linked to high-level functions based on the incarnate system).

Source code files from the CMD function were examined individually to select programs with small, medium, and large numbers of lines of code. This examination was based solely on surface features (e.g., program language, on-line/batch program, program size, and number of files used). The lack of functional structure simplified case study selection because the same program components were likely to be identified regardless of the functional component selected. Ten programs (17 percent of the 59 unit programs) were selected as a reasonable sample size for the case study. The mix of program types was established with the same program percentages of D035A (40 percent batch COBOL, 33 percent IDEAL, and 20 percent CICS COBOL). These percentages are also representative of the makeup of the SC&D system (41 percent batch COBOL, 38 percent

IDEAL, 13 percent CICS COBOL 38 system (41 percent batch COBOL, 38 percent IDEAL, and 13 percent CICS COBOL, and 8 percent other). Application of the D035A percentages to the sample size of ten programs resulted in four batch COBOL programs, four IDEAL programs, and two CICS COBOL programs.

Tables 13 through 15 list the programs selected for the case study and pertinent size information. "Number of functions implemented" refers to the 26 major CMD functions identified in the SC&D functional description extract. "Source listing pages" refers to the number of printed pages of source code in single column, ten pitch format.

Table 13
Batch Programs

Program identification	Number of functions implemented	Lines of code	Source listing pages
ZZLAD057	6	2054	46
ZZLAD058	6	3239	73
ZZLAD513	2	6826	139
ZZLAD555	10	5423	122

Table 14
IDEAL (On-line) Programs

Program identification	Number of functions implemented	Lines of code	Source listing pages
ZZLAI304	19	1719	39
ZZLAI501	1	3413	76
ZZLAI504	7	2203	80
ZZLAI505	10	3581	80

Table 15
CICS COBOL (On-line) Programs

Program identification	Number of functions implemented	Lines of code	Source listing pages
ZZLAI544	3	2505	56
ZZLAI550	1	5426	122

The ten programs (36,389 lines of code) were extracted from the source code in .TXT files and converted to Microsoft Word 6.0 .DOC files. Microsoft Word was used to print source code listings in reduced font size and double column format to reduce the volume of printed material. The line numbering feature was used to add line numbers to IDEAL source programs not normally numbered. The "Find" (search) function was used during program analysis to locate various program elements in source code.

Program ZZLAD057 - Extract MICAP Requisition Data. This batch COBOL program reviews back orders for items needed to return equipment to mission capable (MICAP) status and extracts relevant data for an interfacing system. Multiple documents from the database are retrieved to complete the data extract, and processing is not complicated. The program is executed once in a two step job. Although this program is not functionally related to cataloging management data and should not have been included in the functional area, it was reverse engineered.

Program ZZLAD058 - Extract Interface System Transactions. This batch COBOL program retrieves interface data from interface tables in the database and writes the data

to appropriate output files. Although the program uses 27 different output files, no real processing is performed; data is merely transferred. However, the program is relatively complex because files are conditionally written, dependent on the content of a control record read during the execution of multiple jobs at various points in time. Detailed analysis was not performed on the program because it contains no functional processing. Fourteen provisional functions to produce interface files for other systems were noted for possible inclusion in the domain model.

Program ZZLAD513 - Screen Locally Assigned Stock Number. This batch COBOL program is executed with a series of other programs and performs complicated analysis and manipulation of item interchangeable and substitutable (I&S) data. The program changes existing I&S structure, creates new I&S structure, and generates update transactions for interfacing systems.

Program ZZLAD555 - Preprocess D043 Stock List Changes. This batch COBOL program consists of five major parts and is executed five times from a JCL stream of nine steps. In addition to the five execution steps, there is a step to create a disk file from an input tape and three sort steps. Five control records read by the program during execution determine which of the five parts is performed.

The first part (check label) checks the label of an external system tape to ensure the file has not already been processed. The second part (expand transactions) adds unit price information to stock list changes. The third part (reject duplicates) eliminates duplicate

records from old and new master files, a requirement because stock list changes with effective dates in the future are recycled via the old master file. The fourth part (release records) creates a file of stock list changes ready for release and creates an old master file of changes not ready for release; the old master is then used in the next processing cycle. Part five (monitor transactions) generates JCL to execute and monitor the release of stock list changes via a complicated on-line procedure.

Parts one, three, and five were eliminated from the analysis because they are implementation dependent parts of the program and. The two remaining parts were subjected to detailed analysis (approximately nine percent of the procedure division lines of code).

This program is an example of the kind of difficulties encountered in legacy systems. There is no logical reason to combine five unrelated steps into one complicated program. Although a significant amount of time was required to analyze the program, very little functional information was extracted.

Program ZZLAI304 - Route External System Data. This on-line IDEAL subroutine program receives a transaction image from a calling program, identifies the document type, and passes the transaction to one of 45 subroutines for processing. As no processing is performed by the program, it was eliminated from further analysis.

Program ZZLAI501 - ZAA/ZFA Processor. This on-line IDEAL subroutine program processes new index item record establish transactions (ZAA) and master item establish transactions (ZFA). The subroutine is called from one D035A (SC&D) program and four D035K programs and calls six subroutines. Interface records are created and stored in the database for eventual dispatch to interfacing systems.

Program ZZLAI504 - Process Stock List Change Data, Part II. This on-line IDEAL subroutine program performs specialized processing related to a stock number change. Processing includes posting stock number changes to related records (e.g., backorders, due-out records, usage records, unit of issue, and interchangeable and substitutable stock number data).

Program ZZLAI505 - Stock List Change. This on-line IDEAL subroutine program is one of the main stock list change processing components. The program is called by five programs (four within SC&D, and one from D035K) and conditionally calls 12 other subroutines to post stock list changes to the database. Subroutines called by the program generally perform derivative actions necessitated by stock list changes (e.g., writing data for interfacing systems, requesting item inventories, and processing errors).

Program ZZLAI544 - Print Stock List Change Notice. Despite a relatively simple title, this on-line CICS COBOL program processes stock list change data to update database tables for a shipping system. Comments in the program (unchanged since the program was written) indicate this was originally a batch program within the Shipping Information

System (D035T). The stock list change notices are actually stock list change transactions for distribution to other systems through database interface records. The subroutine is called by three programs and calls two subroutines.

Program ZZLAI550 - Edit Stock List Changes. This on-line CICS COBOL subroutine program is an edit only routine. The program is called by three other programs and calls one subroutine. The program reads 37 database tables, but does not update the database. The program edits various stock list change transactions and returns results to the calling program. Because edit routines are of little value in determining system functions this program was eliminated from further analysis.

Applying the Methodology

Because participation by expert functional users in the development of the domain model was not possible, it was necessary to simulate development using available documentation and general domain knowledge. The domain model was created early in the methodology application to avoid introducing detailed knowledge derived through program reverse analysis. Extracts of the SC&D Functional Description (FD) and the System Specification (SS) for the CMD area were used as source data.

A data collection form was prepared to support initial program analysis. Data collected from individual programs was recorded on the forms and used to enter program data into the database.

Initial program analysis included the collection of identifying information (e.g. program identification, program class and type, language, number of versions, number of authors, source lines of code, size of data and procedure divisions, number of program variables, and number of input and output files). Program header comments were collected when they explained program purpose or processing procedures. Comment line numbers were recorded to facilitate source code reference.

The second level of analysis was program structure review of the ten case study programs. The objective was to develop an index of program complexity to predict the analysis time required for review. A hypothesis was that the most frequently used metric, source lines of code, was not be a suitable estimator for the degree of difficulty in recovering functional design information from source code. The complexity index computed for each program considered the following factors: the number of GO TO, PERFORM, and REDEFINES statements, the number of paragraphs, and the average number of source lines per paragraph; a subjective rating as to program structure, comments, and naming conventions; and a complexity rating calculation that considered the number of input and output files, data and procedures division sizes, and other program average data. The complexity rating calculation was also based on benchmark data reported by COBOL program researchers (i.e., the average number of COBOL data division statements is 1,000; the average number of procedure division statements is 1,200; and the average number of statements in a performed paragraph is 50) (Sneed & Jandrasics, 1987). The

complexity index was used as the basis for an initial prediction of the amount of time required to reverse engineer each program in the test case.

Table 16 summarizes time required for preliminary program analysis.

Table 16
Time Required for Preliminary Program Review

Program name	Initial review time (minutes)	Structure review time (minutes)	Total review time (minutes)
ZZLAD057	20	65	85
ZZLAD058	25	40	65
ZZLAD513	30	45	75
ZZLAD555	35	30	65
ZZLAI304	20	15	35
ZZLAI501	30	15	45
ZZLAI504	22	30	52
ZZLAI505	40	35	75
ZZLAI544	40	25	65
ZZLAI550	35	37	72
Total time	297	337	484
Mean time	29.7	33.7	48.4

The program structure analysis was detailed and designed to support the development of a program model in order to make further source code reference unnecessary. Structure analysis included JCL review for batch programs, preparation of job and job step diagrams for batch programs, calling program diagrams for on-line programs, and input-output data flow diagrams for batch and on-line programs. The results of the JCL analysis are shown in Table 17. (Three non-functional programs are excluded).

Table 17
Time Required for Job Control Language Review

Program name	**JCL/Program call review (minutes)**	**Data flow diagram preparation (minutes)**	**Total review time (minutes)**
ZZLAD057	25	10	35
ZZLAD513	65	20	85
ZZLAD555	463	15	478
ZZLAI501	15	15	30
ZZLAI504	8	20	28
ZZLAI505	40	27	67
ZZLAI544	20	18	38
Total time	636	125	761
Mean time	90.86	17.86	108.71

Other data collected during program structure analysis included calling programs, called programs, input and output files used, database tables used, and a list of non-implementation dependent program paragraphs. Program paragraphs were read to extract functionally-oriented code and to eliminate implementation-oriented code (e.g., paragraphs that open, close, read, and write files; paragraphs that read or write database tables; and paragraphs that edit or validate data). Initially, paragraphs not easily classified as functional or implementation oriented were included to preserve program form or flow. Paragraph header notes were extracted if they contained expanded paragraph names, purpose or procedure information. When additional information could be derived from the paragraph code, "added" notes were also created. Program line numbers for the first lines of notes and paragraphs were recorded to facilitate source code reference. Table 18 summarizes the time required for this part of the program structure analysis.

Table 18
Time Required for Detailed Program Structure Analysis

Program name	Number of source code paragraphs	Number of paragraphs extracted	Total analysis time (minutes)
ZZLAD057	34	15	410
ZZLAD513	113	38	590
ZZLAD555	62	39	1000
ZZLAI501	43	27	335
ZZLAI504	39	39	398
ZZLAI505	50	47	787
ZZLAI544	41	22	313
Totals	382	227	3833
Mean	54.57	32.29	547.57

Because data extracted from the test case programs (especially comments, notes, and paragraph titles) was so extensive, it was impractical to manually produce the physical program models. Relevant tables from the program information database were exported to FoxPro for Windows Version 2.6 to support automatic production of the physical program models. FoxPro was chosen because of the author's previous experience with the FoxPro command language. Changes were made to names because of FoxPro's eight-character table name and ten-character data name restrictions.

The next step was program function analysis--the interpretation of extracted program paragraphs and conversion to functions for eventual assignment to the domain model. Functional analysis was accomplished by extracting a list of paragraph numbers from the database for each test case program. Using the physical program model and the program paragraph listing, each paragraph was evaluated for domain model suitability. Paragraphs

were often "rolled up" to create a higher level abstraction; in only two instances was a program paragraph expanded into more than one function. The paragraph number to function number data was recorded in the data base to allow tracing. Function descriptions were synthesized by interpreting notes associated with program paragraphs, restating them in functional terms, or extracting the meaning of the individual or combined groups of paragraphs. Where the meaning of an extracted paragraph could not be interpreted from the source code, a question mark (?) was inserted. In an actual reverse engineering application, functional users or technicians would be consulted to interpret problem paragraphs.

The final step in the reverse engineering effort was to assign the derived functions to activity "slots" in the domain model, creating upper level subtasks as necessary to maintain balanced decomposition.

Table 19 summarizes the time required for function analysis and process assignment.

Findings

Documentation

The planned primary source of information for the domain model was an extract of the CMD Functional Description (FD). The FD extract was crudely written and poorly structured. Functional process descriptions were generally oriented to physical implementations of the two systems replaced by the modernized SC&D.

Table 19
Time Required for Function Analysis and Process Assignment

Program name	**Paragraphs extracted**	**Paragraphs used**	**Functions developed**	**Analysis time (minutes)**
ZZLAD057	15	14	8	115
ZZLAD513	38	27	16	85
ZZLAD555	39	14	10	100
ZZLAI501	27	21	11	100
ZZLAI504	39	35	14	60
ZZLAI505	47	31	18	48
ZZLAI544	22	13	4	190
Totals	227	155	81	698
Mean	32.29	22.14	11.57	99.71

Descriptions tended to be disjointed narrative that hinted at business rules without clearly specifying them. Functional process descriptions did not match high-level input and output descriptions even when included. Some descriptions mentioned specific data files while others did not. Functions were unbalanced--some were written at high level, others at low level. Many lower-level functions were implied but not clearly specified. Even physical process descriptions were incomplete, stressing certain functional details and omitting others.

Three problems with the FD extract were immediately apparent: (a) the document was written in terms of the processing accomplished by predecessor systems rather than functional requirements to be performed by the modernized system; (b) the functions

were written at such a high level of abstraction that they had meaning only to a functional expert capable of filling in the missing information; and (c) the development of a management information system from the FD was an impossible task.

Even though the FD extract was not an ideal document, the initial domain model was developed by extracting tasks, subtasks, and activities from process descriptions. Intermediate levels were synthesized to introduce a hierarchical decomposition. The completed CMD domain was extremely shallow, consisting of only 28 activities. This was judged to be inadequate for the reverse engineering effort.

The next documentation level in the systems development life cycle, an extract of the System Specification (SS), was used to extend the FD-based domain model. The basic structure of the FD was maintained in the SS. A measure of additional detail and new functions were also added in a few instances. The major improvement to the SS was the addition of "as built" design-oriented information to replace incorrect FD information. Several levels of data flow diagrams supported the SS processing descriptions. Most of the SS process descriptions simply restated the narrative descriptions for equivalent FD functions. The inputs and outputs for SS processes were used to surmise activities.

The final domain model contained 9 tasks, 42 subtasks, and 140 activities, and represented SC&D IMWRP Key Area 6, Cataloging Management Data. While this domain model would not be produced in the same form by a group of functional domain experts during

facilitated modeling sessions, it was considered adequate as a high-level structural framework into which extracted program activities could be placed.

Program Source Code

Program source code was reasonably well structured. Paragraphs tended to be short and performed single, well-defined activities. The paragraph numbering structure facilitated paragraph tracking. However, the logic of several programs could have been broken out into several shorter programs.

Maintenance change comments at the beginning of programs were extensive, as were the number of maintenance changes. Some comments provided clarifying information about terms and document identifiers, but generally they were of limited value to the reverse engineering effort. The major problem was that a change made in one version of a program could be eliminated in a later version of the program. Tracking changes to determine which were not changed again was not considered practical where the reverse engineering objective is to achieve abstraction above the code level as quickly as possible.

Comments in source code were sparse or absent. Generally, each paragraph contained a header comment with a full text title for the COBOL paragraph name. Additional in-line comments, when present, were limited to cryptic notes about an aspect of a paragraph's functions. In-line comments contained a significant number of inaccuracies and errors. For example, code comments were sometimes not changed even when the source code was changed. Substantial domain knowledge was assumed by in-line comment authors

(i.e., document identifier codes, acronyms, and special terms were used freely without explanations). At the end of the test case analysis 227 document identifier codes had been identified while only 123 had been defined.

The use of short data names made program understanding more difficult. Although COBOL permits variable names of up to 30 characters, many names were too short to allow identification. Table 20 provides examples of data names encountered versus the actual name (when known).

Job Control Language

Job control language (JCL) was far more important to batch program reverse engineering than originally believed. In several instances, complex program activity was incomprehensible without lengthy JCL analysis. JCL analysis was performed in three phases: (a) JOB analysis, (b) JCL analysis, and (c) JCL STEP analysis. Job flow diagrams and job step diagrams were prepared for batch programs to augment the source code review. A program calling/called chart was prepared for on-line programs.

Methodology Assessment

Methodology application was relatively straightforward and essentially followed the steps outlined in Chapter III. Steps involving functional and technical users were not performed. If these resources were available, analysis time would have been reduced significantly. For example, considerable analysis time was spent in deciphering acronyms that functional users could have clarified immediately.

Table 20
Example of COBOL Data Names and Equivalent Full-Text Data Names

Program Data Name	Equivalent full-text data name (30 characters or less)
EQP-SPCL-CD	EQUIPMENT-SPECIALIST-CODE
CRIT-ITM-CD	CRITICAL-ITEM-CODE
MNG-DIV-CD	MANAGER-DIVISION-CODE
EAID-CD	?
FC	?
SHLF-LF-CD	SHELF-LIFE-CODE
BGCD	?
REJ-CD	REJECT-CODE
MSN-ESNTL-CODE	MISSION-ESSENTIAL-CODE
WRM-IND	WAR-RESERVE-MATERIEL-INDICATOR
STK-FND-CR-IND	?
DVW-NAME	?
JULI-DATE	JULIAN-DATE
FUNC-CODE	FUNCTION-CODE
RET-CODE	RETURN-CODE
STK-NR	STOCK-NUMBER
F3	?
U-I	UNIT-OF-ISSUE

Some deviation from the methodology was exercised in selecting program paragraphs for domain model inclusion. It was not always possible to definitively categorize a program paragraph as input, output, or transform; therefore, these paragraphs were included in the initial list. Questionable paragraphs were later deleted during the detailed analysis phase when the functional essence was identified.

The three levels of program review (initial analysis, structure analysis, and paragraph analysis) supported incremental acquisition of program knowledge. The approach forced the author to review the broad program scenario before the detailed logic.

Extensive paragraph analysis was not usually required to decipher the function. Database tables and working storage data structures were often adequate to suggest the nature of the processing. Better results would have been achieved if functional and technical personnel had been available to add additional insight.

The most difficult part of the methodology was the interpretation of program paragraphs and the conversion to functions. Although the preceding abstraction process reduced the number of paragraphs to be analyzed, considerable time was required to execute the understand-interpret-translate-describe-assign procedure for each source code paragraph deemed to be a function.

The ability to interpret and translate functions required expert domain knowledge. The unavailability of this knowledge resulted in gaps in the narrative activity descriptions in the final extracted domain model. These gaps, however, were not considered fatal. The extracted domain model represents a useful view of the target subsystem that can be reviewed, refined, and finalized with the participation of domain-intelligent users and technicians.

The process reverse engineering methodology was at times hampered by a lack of detailed data structure knowledge. Limited analysis of database structure was required to understand functional process information; this analysis was performed informally. The program information database contains tables for file record and database table layouts,

but these were not used during the test case application. A parallel data reverse engineering methodology would have improved the process reverse engineering effort.

Comparison of User and Derived Function Models

It was assumed prior to beginning the investigation that available documentation would support an assessment of the extracted domain model. However, the test case indicated that the results of the domain modeling exercise as augmented by the reverse engineering of program activities resulted in a functional process description infinitely more detailed, understandable, and useful than existing system documentation. Another objective of the case study was to use the domain model to validate the methodology. However, the existing domain model in the FD was so shallow that it was not suitable for methodology validation.

Two hundred twenty-seven paragraphs were initially extracted from the 382 test case program paragraphs. Seventy-two of these paragraphs were deleted during detailed program function analysis because they did not equate to functional activities. Many were upper-level program structure paragraphs that called lower level functions. These "duplicate" paragraphs were not immediately apparent during initial paragraph analysis, particularly when lower-level perform statements were located in upper-level paragraph bodies.

Table 21 summarizes information relative to paragraphs extracted from programs.

Table 21
Paragraphs Extracted from Programs and Used in the Domain Model

Program name	Number of paragraphs	Paragraphs selected	Paragraphs used	Percent selected	Percent used
ZZLAD057	34	15	14	44.1	41.2
ZZLAD513	113	38	27	33.6	23.9
ZZLAD555	62	39	14	62.9	22.6
ZZLAI501	43	27	21	62.8	48.8
ZZLAI504	39	39	35	100.0	89.7
ZZLAI505	50	47	31	94.0	62.0
ZZLAI544	41	22	13	53.7	31.7
Totals	382	227	155	59.4	40.6

Table 22 shows the disposition of the 228 extracted paragraphs after the paragraph-to-domain model function allocation. The majority of the program paragraphs were assigned to a single domain model activity. In only two cases did a paragraph break out into more than one domain model activity; these two paragraphs are not included in the total in the table.

Metrics

In a legacy system reverse engineering project, a fundamental problem is estimating the amount of work and time involved. Therefore, a secondary objective of this research was to develop predictor metrics to support an estimation of the amount of time required to reverse engineer a program.

Table 22
Number of Program Paragraphs Allocated to Functions

Number of functions	Number of paragraphs in function	Total paragraphs
53	1	53
9	2	18
5	3	15
2	4	8
3	5	15
0	6	0
1	7	7
1	8	8
1	9	9
1	10	10
0	11	0
1	12	12
Total		155

Candidate factors for these predictor metrics included source lines of code (SLOC), procedure division lines of code (PDLOC), number of procedure division paragraphs (NOPARA), and complexity index (CI).

SLOC is the "standard" metric for estimating the amount of work related to software development or software maintenance. Although SLOC is not always a good predictor, it is understood and commonly used. PDLOC, a variation of SLOC, emphasizes the program procedure division--the area of primary interest for process reverse engineering. NOPARA, also a variation of SLOC, focuses on the lowest structure level addressed by the methodology. Developed as a part of this research, the complexity index (CI) quantifies multiple factors that may contribute to the difficulty in reverse engineering

legacy system source code. The search for predictor metrics focused on these four factors.

Table 23 summarizes test case program analysis time for various phases of the reverse engineering effort. Three programs were not analyzed beyond the initial review stage because they were implementation oriented and did not contain functional processes. An exception is Program ZZLAD555: although only 40 percent of this program is functionally oriented, an an inordinate amount of time was spent in discovering and understanding the processing performed. In a normal reverse engineering environment, much less time would have been spent on the program after the relevant portions were identified. To compensate for the extra, non-productive effort, only 40 percent of the actual analysis time is recorded in the table for the last two activities (DPSA and PFA). Other table entries are reasonably accurate measures of elapsed clock time recorded as the methodology steps were executed.

The analysis times for initial program review (IPR), program structure review (PSR), job control language (JCL) review, and data flow diagram(DFD)/calling chart preparation were relatively constant. However, JCL mean analysis time was increased significantly by the time spent on Program ZZLAD555. If ZZLAD555 is excluded from the calculation, the mean and standard deviation for this value are 29 minutes and 18 minutes respectively (these values are shown in parentheses in Table 23). Preliminary analysis steps

Table 23
Summary of Program Review and Analysis Times

Program ID	IPR	PSR	JCL	DFD	DPSA	PFA	Total	SLOC	PDLOC	NOPARA	CI
ZZLAD057	20	65	25	10	400	115	645	2054	1267	34	91
ZZLAD058	25	40	-	-	-	-	-	-	-	-	355
ZZLAD513	30	45	65	20	590	85	835	6826	4020	113	285
ZZLAD555	35	30	185	15	400	100	765	5423	3493	62	254
ZZLAI304	20	15	-	-	-	-	-	-	-	-	126
ZZLAI501	30	15	15	15	335	100	510	3413	2110	43	208
ZZLAI504	22	30	8	20	398	60	538	2203	1542	39	124
ZZLAI505	40	35	40	27	787	48	977	3581	2141	50	279
ZZLAI544	40	25	20	18	313	190	606	2505	1487	41	53
ZZLAI550	35	37	-	-	-	-	-	-	-	-	145
Mean	30	34	51 (29)	18	462	100	697				
Standard deviation	7	14	57 (18)	5	156	43	170				

Note: IPR=initial program review, PSR=program structure review, JCL=job control language (batch) or program call chart (on-line), DPSA=detailed program structure analysis, PFA=paragraph function analysis; mean and standard deviation rounded to nearest whole minute. The times shown do not include database data entry time.

represented a small amount of overall analysis time and could be estimated using the mean and standard deviations.

Detailed program structure analysis (DPSA) and program function analysis (PFA) represented the major part of the reverse engineering effort. Since SLOC, PDLOC, NOPARA, and CI are also presumed to influence reverse engineering time, a direct linear relationship was hypothesized.

Scatterplots of SLOC, PDLOC, NOPARA, and CI values and total reverse engineering times were prepared to identify possible linear trends (see Figures 29-32). Although the

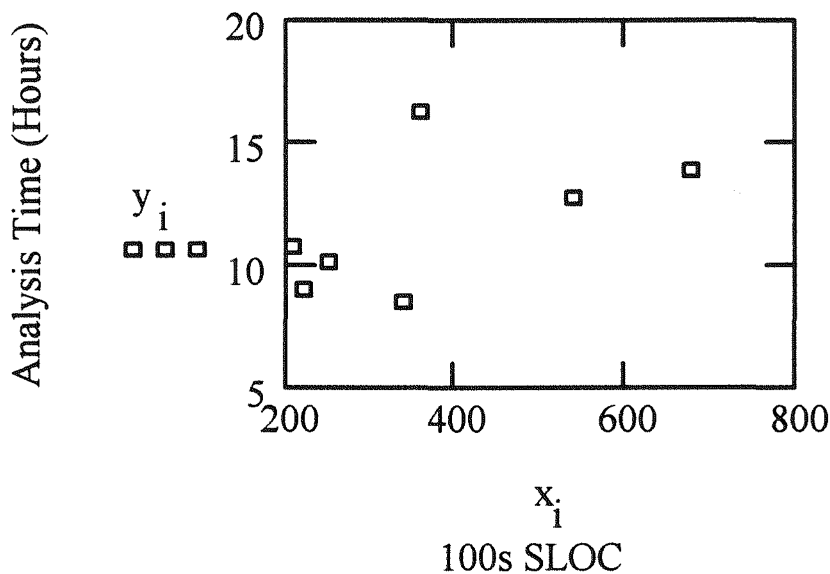


Figure 29. SLOC scatterplot.

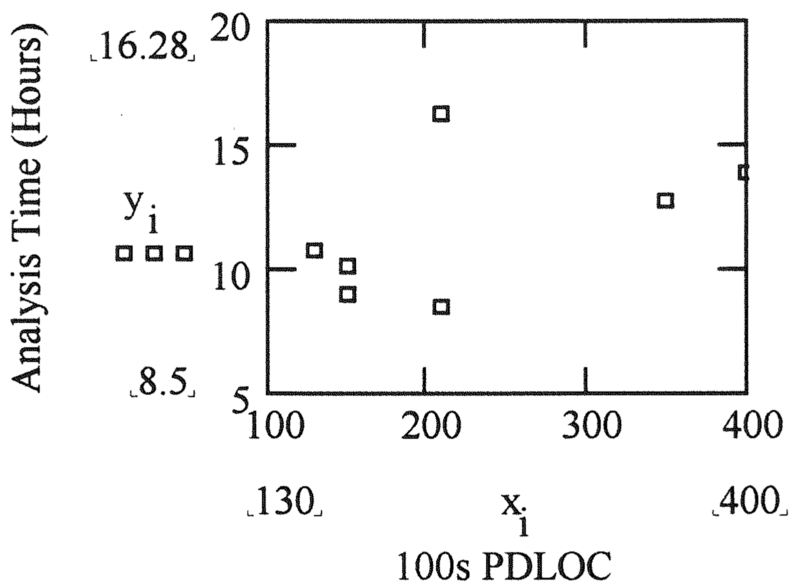


Figure 30. PDLOC scatterplot.

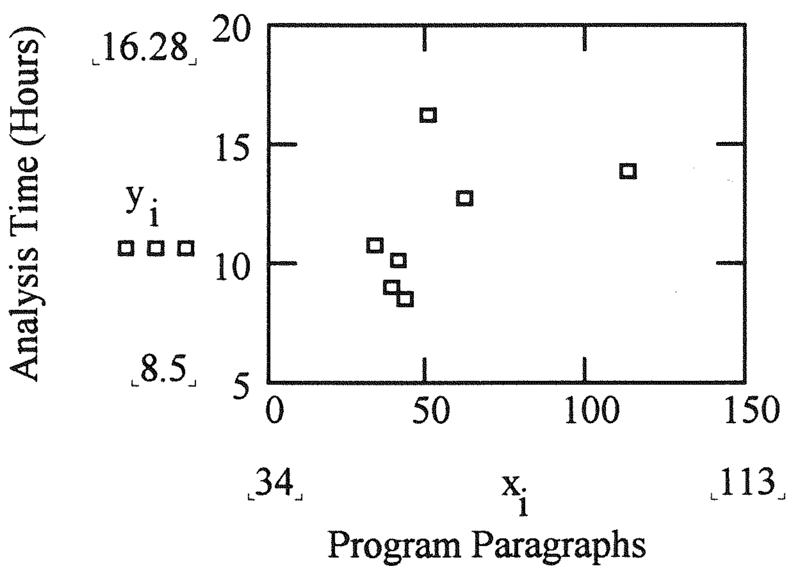


Figure 31. NOPARA scatterplot.

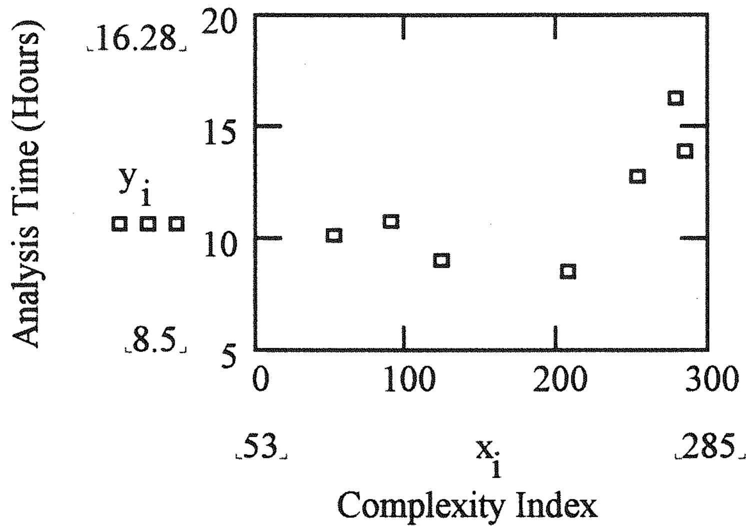


Figure 32. CI scatterplot.

number of data points is small, each of the plots showed an overall trend toward linearity. Therefore, linear regression was selected as the statistical analysis method for development of predictor metrics.

Source Lines of Code

Source lines of code (SLOC) were rounded to the nearest 100 lines and identified as the regression analysis x-axis (predictor) variable. Total analysis time was converted to hours and fractions of hours and identified as the regression analysis y-axis (response) variable. The computed regression line formula was determined to be y = 8.3043 + 0.0089x where y is the estimate of total analysis time required in hours, 8.3043 is the y-intercept, and 0.0089 is the slope of the regression line. The data plots and regression line are shown in Figure 33.

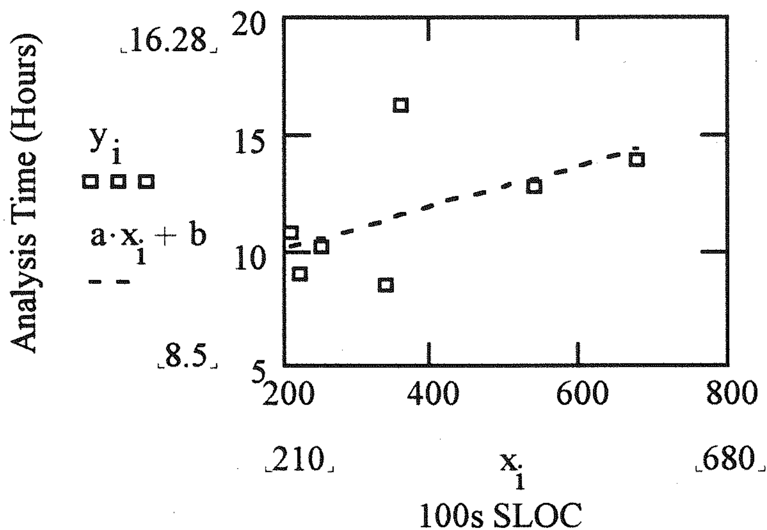


Figure 33. Regression line SLOC.

The regression line was checked by showing the means (x = 371.4286, y = 11.61) are on the regression line (y = 8.3043 + 0.0089 (371.4286) = 11.61). The y-intercept equals 8 hours and 18 minutes. The variance of the regression line is 8.0176 and the standard deviation is 2.8315. A predicted value of y for an x value within the range of x data values analyzed should be within plus or minus one standard deviation 95 percent of the time. Table 24 is the analysis of variance (ANOVA) table for the SLOC linear regression analysis.

Table 24
ANOVA for the SLOC Linear Regression Analysis

Source of variation	Degrees of freedom	Sum of squares	Mean of squares
Regression	1	14.8194	14.8194
Error	n - 2	33.2864	6.6573
Total	6	48.1058	

If there is a relationship between SLOC and reverse engineering analysis time, the slope of the regression function is not equal to zero. Therefore, the null and alternate hypotheses are:

H_o: $\beta_1 = 0$

H_1: $\beta_1 \neq 0$

The null hypothesis is rejected if F* is greater than $F_{.05}\ \{1, n - 2\} = 6.61$. AS F* = 2.2260 is not greater than 6.61, the null hypothesis is rejected and it is concluded that $\beta_1. \neq 0$. It is concluded there is a linear association between the number of source lines of code (in 100s) and reverse engineering analysis time.

The regression standard error estimate is 2.5802. The coefficient of linear determination is 0.3081; the coefficient of linear correlation is 0.5550. As residual points do not appear to be randomly scattered above and below the horizontal axis, the assumption of linearity is questionable (see Figure 34). Tests of equal variance or normality are not possible because there are not enough data points, but the plotted points do not appear to be equally spread. Since $S_e = 2.5802$, all of the residuals have absolute values less than $2S_e$,

indicating no deviation from normality. As the residual for 3,600 SLOC (4.77) is within $2S_e$, it is not considered an outlier.

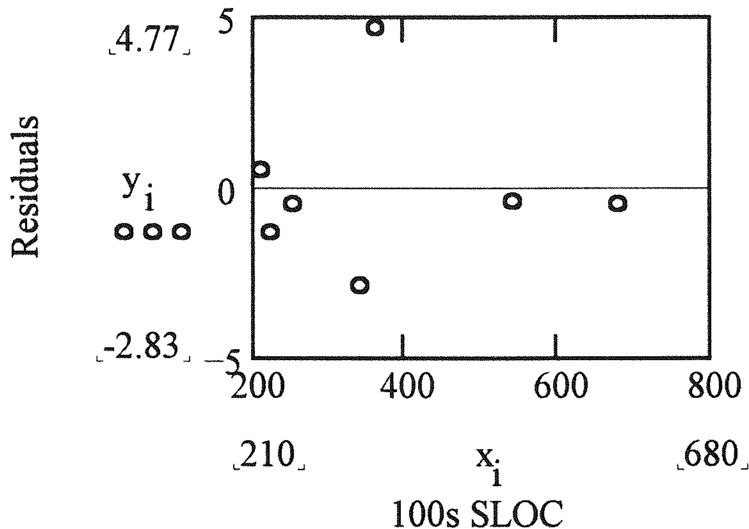


Figure 34. Residuals plot SLOC.

Confidence intervals for the slope of the regression line at the 95 percent level are –0.0064 to 0.2421. Therefore, the predicted reverse engineering analysis time for a 3,000 line program at the 95 percent confidence level is:

Low value: $y = 8.3043 + (-0.0064)\ (300) = 6.3843$

High value: $y = 8.3043 + 0.0242\ (300) = 15.5643$

The value predicted from the regression line is $y = 8.3043 + 0.0089\ (300) = 10.9743$. It is estimated that reverse engineering a 3,000-line program will take between 6 hours and 24

minutes and 15 hours and 36 minutes. A prediction derived directly from the regression formula is approximately 11 hours.

Procedure Division Lines of Code

Procedure division lines of code (PDLOC) were rounded to the nearest 100 lines and identified as the regression analysis x-axis (predictor) variable. Total analysis time was converted to hours and fractions of hours and identified as the regression analysis y-axis (response) variable. The computed regression line formula was determined to be $y = 8.4786 + 0.0137x$, where y is the estimate of total analysis time required in hours, 8.4786 is the y-intercept, and 0.0137 is the slope of the regression line. The data plots and the regression line are shown in Figure 35.

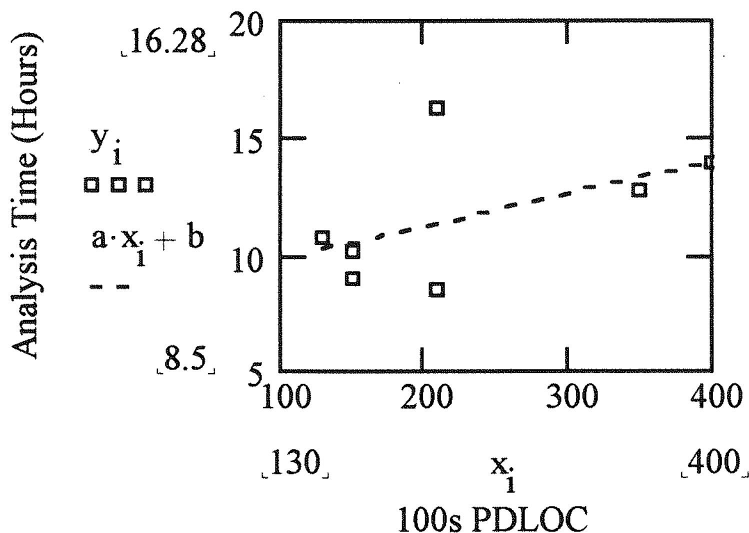


Figure 35. Regression line PDLOC.

The regression line was verified by showing that the means (x = 228.5714, y = 11.61) are on the regression line (y = 8.4786 + 0.0137 (228.5714) = 11.61). The y-intercept equals 8 hours and 15 minutes. The variance of the regression line is 8.0176 and the standard deviation is 2.8315. A predicted value of y for a value of x in the range of data analyzed should be within plus or minus one standard deviation 95 percent of the time. Table 25 is the analysis of variance (ANOVA) table for the PDLOC linear regression analysis.

Table 25
ANOVA for the PDLOC Linear Regression Analysis

Source of variation	Degrees of freedom	Sum of squares	Mean of squares
Regression	1	12.6355	12.6355
Error	n - 2	35.4703	7.0940
Total	6	48.1058	

The PDLOC regression analysis null and alternate hypotheses are the same as the SLOC hypotheses:

H_o: $\beta_1 = 0$

H_1: $\beta_1 \neq 0$

As F* = (1.7040) is not greater than 6.61, the null hypothesis is rejected and it is concluded that β_1. $\neq 0$. It is concluded there is a linear association between the number of procedure division lines of code (in 100s) and reverse engineering analysis time.

The regression standard error of the estimate is 2.6635. The coefficient of linear determinations is 0.2627; the coefficient of linear correlation is 0.5125. As residual points do not appear to be randomly scattered above and below the horizontal axis, the assumption of linearity is questionable (see Figure 36). Tests of equal variance or

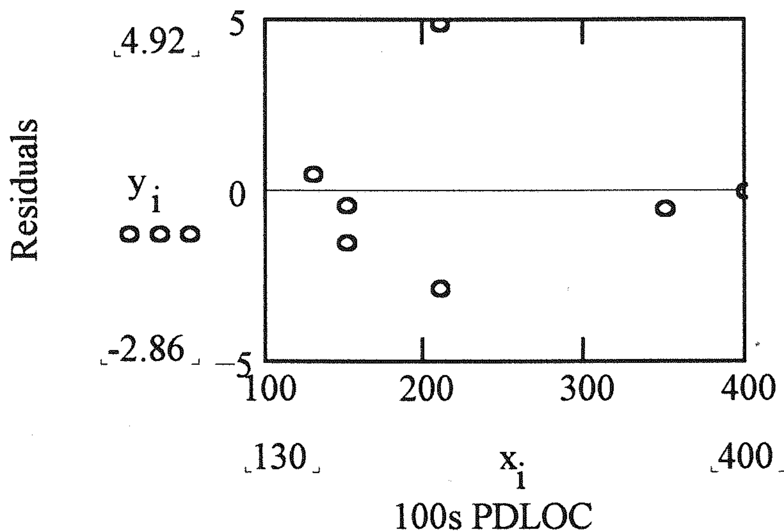


Figure 36. Residuals plot PDLOC.

normality are not possible because there are not enough data points, but the plotted points do not appear to be equally spread. Since $S_e = 2.6635$, all of the residuals have absolute values less than $2S_e$, indicating no deviation from normality. As the residual for 2,100 PDLOC (4.92) is within $2S_e$, it is not considered an outlier.

Confidence intervals for the slope of the regression line at the 95 percent level are −0.0128 to 0.0402. Therefore, the predicted reverse engineering analysis time for a program with 1,500 procedure division lines of code at the 95 percent confidence level is:

Low value: $y = 8.4786 + (-0.0128)(150) = 6.5586$

High value: $y = 8.4786 + 0.2421(150) = 14.5086$

The value predicted from the regression line is $y = 8.4786 + 0.0137(150) = 10.5536$. It is estimated that reverse engineering a program with a 1,500 lines of procedure division code will take between 6 hours and 36 minutes and 14 hours and 36 minutes. A prediction derived directly from the regression formula is 10 hours and 30 minutes.

Number of Procedure Division Paragraphs

The number of procedure division paragraphs (NOPARA) in each program was used as the regression analysis x-axis (predictor) variable. Total analysis time was converted to hours and fractions of hours and identified as the regression analysis y-axis (response) variable. The computed regression line formula was determined to be $y = 8.7128 + 0.0531x$, where y is the estimate of total analysis time required in hours, 8.7128 is the y-intercept, and 0.0531 is the slope of the regression line. The data plots and the regression line are shown in Figure 37.

The regression line was verified by showing that the means ($x = 54.5714$, $y = 11.61$) are on the regression line ($y = 8.7128 + 0.0531(54.5714) = 11.61$). The y-intercept equals to 8 hours and 42 minutes. The variance of the regression line is 8.0176; the standard deviation is 2.8315. A predicted value of y for a value of x within the range of data analyzed should be within plus or minus one standard deviation 95 percent of the time.

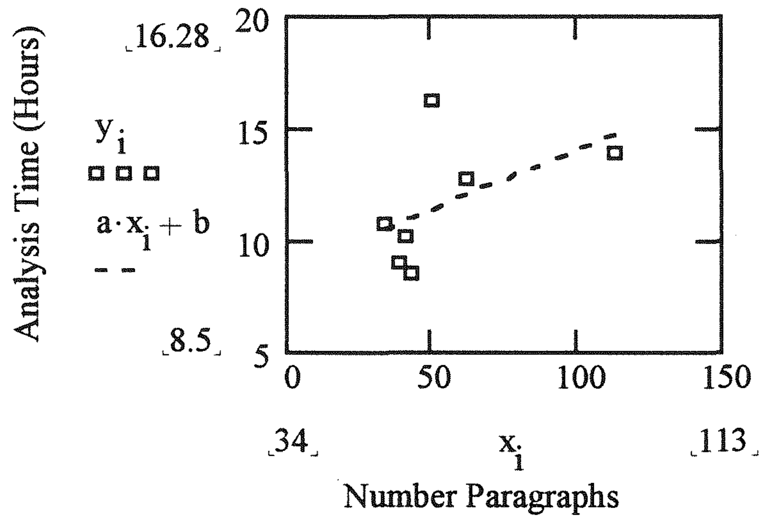


Figure 37. NOPARA regression line.

Table 26 is the analysis of variance (ANOVA) table for the NOPARA linear regression analysis.

Table 26
ANOVA Table for the NOPARA Linear Regression Analysis

Source of variation	Degrees of freedom	Sum of squares	Mean of squares
Regression	1	12.4946	12.4946
Error	n - 2	35.6112	7.1022
Total	6	48.1058	

The NOPARA regression analysis null and alternate hypotheses are the same as the SLOC hypotheses:

H_o: $\beta_1 = 0$

H_1: $\beta_1 \neq 0$

As F* = (1.7543) is not greater than 6.61, the null hypothesis is rejected and it is concluded that $\beta_1. \neq 0$. It is concluded there is a linear association between the number of program paragraphs and reverse engineering analysis time.

The regression standard error estimate is 2.6687. The coefficient of linear determination is 0.2957; the coefficient of linear correlation is 0.5116. As residual points do not appear to be randomly scattered above and below the horizontal axis, the assumption of linearity is questionable (see Figure 38). Tests of equal variance or normality are not possible because there are not enough data points, but the plotted points do not appear to be equally spread. Since $S_e = 2.6687$, all of the residuals have absolute values less than $2S_e$, indicating no deviation from normality. As the residual for 50 paragraphs (4.91) is within $2S_e$,, it is not considered an outlier.

Confidence intervals for the slope of the regression line at the 95 percent level are −0.0495 to 0.1557. Therefore, the predicted reverse engineering analysis time for a program consisting of 40 paragraphs at the 95 percent confidence level is:

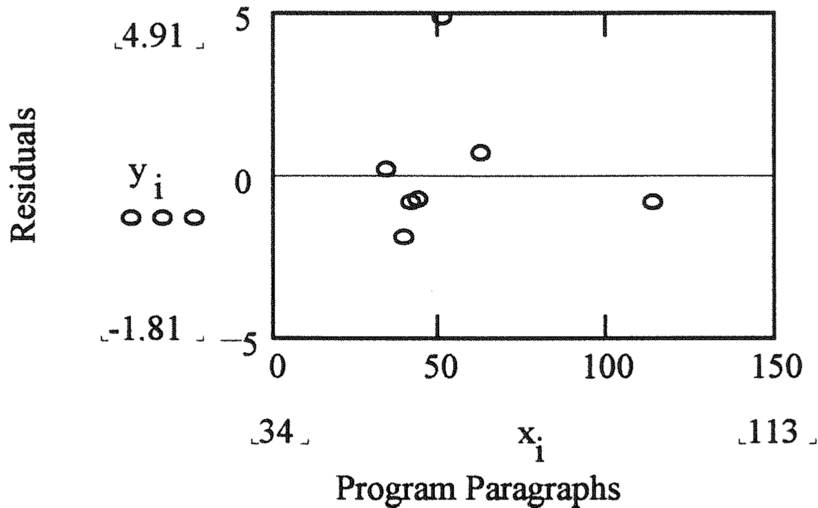


Figure 38. NOPARA residuals plot.

Low value: y = 8.7128 + (–0.0495) (40) = 6.7328

High value: y = 8. 7128 + 0. 0.1557 (40) = 15.0208

The value predicted from the regression line is y = 8.7128 + 0.0531 (40) = 10.8368. It is estimated that reverse engineering a 40 paragraph program will take between 6 hours and 42 minutes and 15 hours. A prediction derived directly from the regression formula is 10 hours and 48 minutes.

Complexity Index

The computed program complexity index (CI) was used as the regression analysis x-axis (predictor) variable. Total analysis time was converted to hours and fractions of hours and identified as the regression analysis y-axis (response) variable. The computed regression line formula was determined to be y = 7.7835 + 0.0207x, where y is the

estimate of total analysis time required in hours, 7.7835 is the y-intercept, and 0.0207 is the slope of the regression line. The data plots and the regression line are shown in Figure 39.

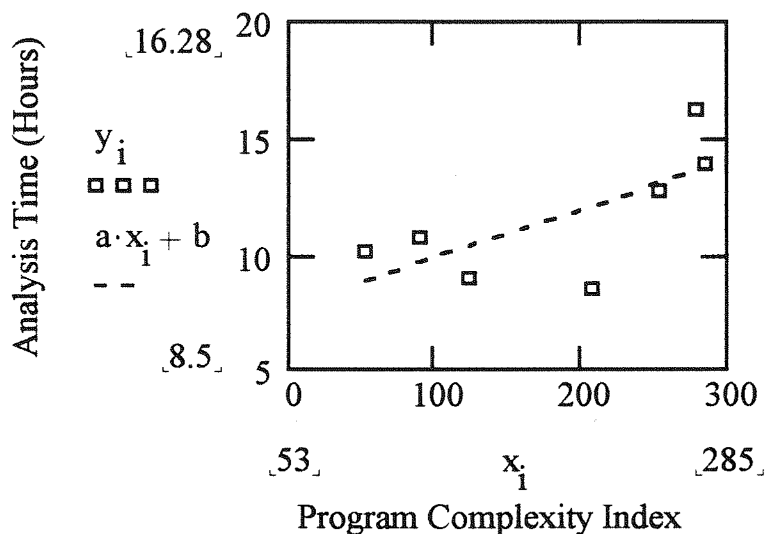


Figure 39. CI regression line.

The regression line was verified by showing that the means (x = 184.8571, y = 11.61) are on the regression line (y = 7.7835 + 0.0207 (184.8571) = 11.61). The y-intercept equals 7 hours and 48 minutes. The variance of the regression line is 8.0176; the standard deviation is 2.8315. A predicted value of y for a value of x within the range of data analyzed should be within plus or minus one standard deviation 95 percent of the time. Table 27 is the analysis of variance (ANOVA) table for the CI linear regression analysis.

Table 27
ANOVA for the CI Linear Regression Analysis

Source of variation	Degrees of freedom	Sum of squares	Mean of squares
Regression	1	23.1138	23.1138
Error	n - 2	24.9910	4.9983
Total	6	48.1058	

The CI regression analysis null and alternate hypotheses are the same as the SLOC hypotheses:

H_o: $\beta_1 = 0$

H_1: $\beta_1 \neq 0$

As F* = (4.6243) is not greater than 6.61, the null hypothesis is rejected and it is concluded that $\beta_1. \neq 0$. It is concluded there is a linear association between program CI and reverse engineering analysis time.

The regression standard error estimate is 2.2357. The coefficient of linear determination is 0.4804; the coefficient of linear correlation is 0.6932. As residual points seem to be randomly scattered above and below the horizontal axis, the assumption of linearity appears to be met (see Figure 40). Tests of equal variance or normality are not possible because there are not enough data points, but the plotted points are nearly equally spread. Since $S_e = 2.3575$, all of the residuals have absolute values less than $2S_e$, indicating no deviation from normality. As the residuals for CI values 208 and 279 (-3.59 and 2.72 respectively) are within $2S_{e,}$, they are not considered outliers.

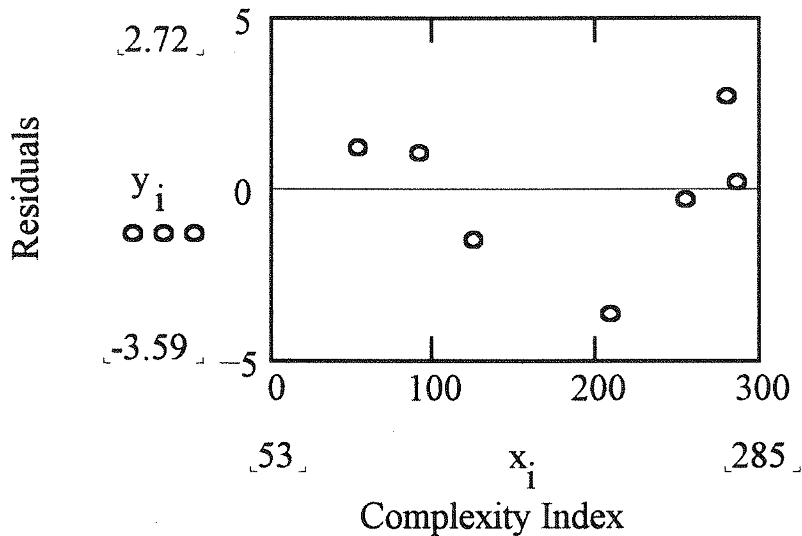


Figure 40. Residuals plot CI.

Confidence intervals for the slope of the regression line at the 95 percent level are −0.0040 to 0.0454. Therefore the predicted reverse engineering analysis time for a program with a CI of 150 at the 95 percent confidence level is:

Low value: $y = 7.7835 + (-0.0040)\,(150) = 8.3835$

High value: $y = 7.7835 + 0.0454\,(150) = 14.5935$

The value predicted from the regression line is $y = 7.7835 + 0.0207\,(150) = 10.8834$. It is estimated that reverse engineering a program with a CI of 150 will take between 8 hours and 24 minutes and 14 hours and 36 minutes. A prediction derived directly from the regression formula is 10 hours and 54 minutes.

Metrics Analysis Summary

Table 28 summarizes the results of the regression analysis. Note that calculations producing variance (8.0176) and standard deviation (2.8315) are based on the y-axis variable (reverse engineering analysis time). These values are the same for all factors and are not included in the table.

Table 28
Summary of Regression Analysis Results

Factor	Intercept/ Slope	Range of 95% CI	Standard error	Linearity coefficient	Linear correlation coefficient
SLOC	8.3 0.0089	0.0640 to 0.2421	2.5802	0.3081	0.5550
PDLOC	8.5 0.0137	− 0.1280 to 0.0402	2.6635	0.2627	0.5125
NOPARA	8.7 0.0531	− 0.0495 to 0.1557	2.6687	0.2957	0.5116
CI	7.8 0.0207	− 0.0004 to 0.0454	2.2357	0.4804	0.6932

Based on the results of the analysis, it is concluded that the source lines of code (SLOC) and the complexity index (CI) are reasonable prediction variables for estimating the time required to reverse engineer a COBOL program. Of the two factors, CI has the lowest S_e, the highest coefficients of linear determination and linear correlation, and the narrowest 95 percent confidence intervals for the regression line slope.

SLOC is a suitable predictor value because it is available from source code listings without detailed analysis. CI is also a suitable predictor value, but requires detailed program analysis before assigning the CI rating. While SLOC can be used to establish the initial estimate of reverse engineering time, CI can be used to make a refined estimate after completing program structure review.

At the beginning of the case study, "best guess" estimates of the time required to reverse engineer each of the seven programs were made based on program size. Table 29 compares initial estimates, actual time, and time projected using the SLOC and CI regression formulas.

Table 29
Comparison of Initial, Actual, and Computed Reverse Engineering Times

Program name	SLOC (100s)	CI	Initial estimate (hours)	Actual time (hours)	CI estimate (hours)	SLOC estimate (hours)
ZZLAD057	210	91	9.0	10.8	9.7	10.2
ZZLAD513	680	285	28.0	13.9	13.7	14.4
ZZLAD555	540	254	16.0	12.8	13.0	13.1
ZZLAI501	340	208	11.0	8.5	12.1	11.3
ZZLAI504	220	124	10.0	9.0	10.4	10.3
ZZLAI505	360	279	13.0	16.3	13.6	11.5
ZZLAI544	250	53	10.0	10.1	8.9	10.3

The SLOC and CI regression formulas can only be used for values that fall between 2,100 and 6,800 lines of code and for complexity indices that fall between 53 and 285 (i.e., within the observed ranges on which the regression analysis was based). Of the 59 programs of the CMD subsystem, at least 30 fall within the SLOC ranges. The SLOC

estimator could therefore be used to project reverse engineering time for these programs. Programs with SLOC and CI outside these ranges must be reverse engineered and the results used to update the regression analysis in order to provide a broader estimation range.

Methodology Changes

The major change to the methodology was the additional analysis of batch program JCL. The process used in the test case was:

1. JCL listings were searched for STEP statements identifying test case programs (e.g., //STEP010 EXEC PGM=ZZLAD057).
2. Identification of the JCL stream was extracted from the JCL header (e.g., ZZJAD202).
3. JOB listings were searched for occurrences of the JCL header name, with a P replacing the J (e.g., ZZJAD202 = ZZPAD202).
4. JOB and JCL listings were printed and placed with program listings.

The JOB stream comments often contained a short description of the program and identification of the files produced. File notes did not indicate which programs in a series created the files; this information was obtained from individual STEP DD statements. The JOB and STEP JCL statements were used to prepare job flow diagrams and input/output data flow diagrams. In a full-scale application of the methodology, the sequence would be reversed to identify individual programs from JOB statements.

In addition to showing methodology feasibility, the test case was used to evaluate the adequacy of the program information database. Table changes were:

1. Component (Table 4) - A six position numeric attribute, Avg-Data-Name-Length, was added.
2. Function (Table 17) - Text-Function was changed to a Memo field.
3. Narrative (Table 19) - The table was deleted. A narrative field was contained in Function.
4. Narrative-Function (Table 20) - The table was deleted.
5. Record (Table 29) - A 30-character attribute, Record-Name; was added. Layout-Record-Text was deleted.
6. Table (Table 32) - Name-Table-Actual and Table-Prefix were deleted. An eight-character attribute, Data-View-Name, was added.

Two other database changes were identified but not implemented: (a) the line numbers of CALL, LINK, or EXECUTE CICS LINK statements should be recorded in the Component-Component table (Table 6) to facilitate statement location during detailed program analysis, and (b) a one-character field should be added to the Paragraph table (Table 23) to record the disposition of a paragraph selected from a program (X = not used, M = paragraph combined with at least one other paragraph to form a single function, S = paragraph assigned to a single function, and C = paragraph was split into more than one function).

Summary of Results

The process-oriented reverse engineering methodology was demonstrated to be both feasible and practical in recovering functional information from legacy system COBOL source code. The steps outlined in the methodology can simplify the complex activity of converting program code to functional information. The test case also demonstrated the critical importance of involving knowledgeable functional people in reverse engineering activities, during both the preparation of a high-level domain model and the interpretation and assignment of extracted program paragraphs to domain model activities.

Difficulties in understanding programs because of limited data structure knowledge confirmed the belief that a comprehensive reverse engineering methodology must include a data structure recovery component to complement the process recovery component.

The program information database implemented to support the test case was useful in collecting and managing the data related to programs and functions. The program implementation model automatically produced from the database after completion of program structure analysis dramatically reduced the need to refer to program source code.

Although not produced during the test case application, the functional process model (domain model) could have been maintained in the database and automatically printed.

Elements of the domain model stored in a hierarchical structure could easily be extracted at various detail levels to support documentation efforts or to produce functional requirement documents for a system replacement project.

Two possible metrics for predicting work effort associated with reverse engineering were isolated and shown to produce reasonable projections. Although source lines of code (SLOC) was thought to be an inaccurate estimator, a reasonable linear relationship was shown to exist. A complexity index (CI) computed using information about individual programs was also shown to have a linear relationship with reverse engineering time. The SLOC regression formula can be used to estimate the effort required to reverse engineer a system by calculating estimated time for individual programs in the system. The advantage of this predictor is that it can be found easily by counting lines of code and without any other program analysis. The CI predictor requires more detailed program knowledge, but can be used to refine initial program reverse engineering time estimates based on SLOC.

Confidence intervals at the 95 percent level for both SLOC and CI were also computed.

Although the sample size used in the test case was small, there is a reasonable expectation that the results will be corroborated with the application of the methodology to an actual system.

Chapter V

Conclusions

Conclusions

The six objectives established for the investigation in Chapter I were satisfied with the design and application of the process reverse engineering methodology described in Chapters III and IV. These objectives were:

1. Develop a useful, applied approach to high-level design information recovery.
2. Support the validity of the approach by reference to relevant theory.
3. Demonstrate methodology feasibility by using a case study.
4. Demonstrate methodology utility by using a case study.
5. Assess the approach for practical application.
6. Form a foundation for future research.

A comprehensive review of the literature and research in the areas of software forward engineering, reengineering, maintenance, programming languages, program understanding, and reengineering and reverse engineering tools provided the foundation for developing a process reverse engineering methodology. To limit the scope of the investigation, data reverse engineering was excluded. The methodology was designed to recover functional design information from legacy system COBOL source code in the Air Force logistics

systems environment; it was developed by combining top-down information engineering techniques with conventional bottom-up program analysis techniques. The top-down component was designed to replace domain knowledge that was lost during forward engineering. The bottom-up component was designed to identify and separate implementation dependent program components from functional components in order to reduce the amount of source code requiring interpretation.

The methodology was successfully applied to a test case composed of actual programs, demonstrating the feasibility and practicality of the approach. A prototype program information database as constructed and populated to support the methodology. The database was extremely useful in recording and manipulating the large amount of test case program information. Using the methodology, recovered functional information was more accurate, detailed, and useful than the original formal system documentation.

The methodology is detailed in its approach and can be used to train junior analysts. Properly trained junior analysts can perform much of the up-front reverse engineering work, while the critical part of the methodology (i.e., interpreting source code paragraphs and developing domain model functions) requires more experience. Experienced analysts can then perform the more difficult aspects of the reverse engineering methodology.

Although difficult to verify, the methodology is believed to be more efficient than an unstructured "brute force" approach to reverse engineering because it provides for

planning, measurement, and control and separates the process into several distinct phases, each with well-defined outputs.

As program source lines of code were not believed to be a satisfactory indicator of reverse engineering difficulty, the following factors were evaluated as possible predictive metrics: procedure division source lines of code, number of program paragraphs and a program complexity index. Linear relationships were revealed between all factors and reverse engineering analysis time. Two metrics, source lines of code and the program complexity index, were found to be more accurate predictors. Source lines of code were judged to be reasonably accurate for initial estimates of program reverse engineering time. The complexity index, which requires more detailed program analysis to compute, was judged to be more accurate for revised estimates.

To keep the investigation at an achievable level, the reverse engineering methodology was limited to a single application domain (military logistics), a single programming language (COBOL), an IBM MVS/CICS operating system environment, and a small case study. Results may differ in other application domains or environments, or with a larger test case.

The number of reverse engineered programs was smaller than anticipated--three programs determined to be non-functional were discarded after initial analysis. In addition, the test case programs did not include all of the features addressed in the methodology (e.g., on-line screens, and CICS files). Therefore, some components of the methodology were not exercised.

A major constraint was the unavailability of functional area specialists to supply domain knowledge during methodology application. The literature review clearly showed that missing domain knowledge in source code was the principal impediment to reverse engineering. At the end of the test case, some activities remained undefined or speculative. These results were not considered a reflection of a weakness in the methodology, however, because the extracted domain model must be validated by the functional user community before it can be considered complete.

Implications

The proposed methodology is believed to be the first practical, start-to-finish process reverse engineering approach to be described. Unlike research projects focused on automatic reverse engineering methods seldom suitable for practical use, the methodology was designed for application by working reverse engineers. The utility of the methodology was demonstrated by its application to a test case of nearly 40,000 lines of source code.

Reverse engineering is so tightly coupled with human intelligence that current artificial intelligence and knowledge-based techniques are not able to automatically reverse engineer source code. For reverse engineering to be successful, functional domain experts must provide missing domain knowledge.

Reverse engineering techniques must be tailored to specific environments. Different operating systems, programming languages, file structures, and database management systems will require minor methodology modifications. The fundamental concepts of the methodology, however, are applicable to a variety of environments.

Reverse engineering case studies of large legacy systems are rare. This investigation contributes to the information systems field by presenting the results of a formal case study. Sufficient case study documentation allows the research to be duplicated for verification purposes or to extend the methodology.

The major contribution of this research is a new approach to reverse engineering that recognizes the critical importance of lost domain knowledge. The creation of a structured domain model as a preliminary reverse engineering activity before source code analysis is a new reverse engineering approach.

Recommendations

Process reverse engineering for design information recovery offers many opportunities for further study:

1. Applying the methodology to a small system to further validate results.
2. Implementing the manual methodology as a computer-based tool to enhance its effectiveness in reverse engineering large systems. A computer-based source code scanner capable of performing initial analysis and program structure review appears to be feasible and could significantly reduce program analysis time. Providing the ability

to scan source code paragraphs in order to highlight significant information and automatically enter it into a database could further reduce analysis time.

3. Testing the teachability of the methodology by designing an experiment wherein two groups of reverse engineers attempt to recover design information from a small system. One group would be given formal training in the methodology, the other would not. The reconstructed designs and the amount of time required to complete the designs would be compared to determine if learning took place and to demonstrate the effectiveness of the methodology training.
4. Completing the methodology by incorporating a data reverse engineering component. The data reverse engineering component could be satisfied by integrating an existing data modeling methodology into the process methodology or by applying the information engineering approach to methodology design.
5. Refining the source lines of code and complexity index regression models with additional data to improve their value as reverse engineering predictor values. This refinement could be combined with larger test cases from different domains.

Summary

The extent of the problem of aging information systems is reflected in the estimate that there are 100 billion lines of legacy source code worldwide, 80 percent written in COBOL. In the United States alone, approximately $30 billion per year is spent on maintaining legacy system code. More people are maintaining legacy system code than developing new code.

Investment in legacy system software is substantial in terms of original development costs, long-term maintenance costs, and embedded business knowledge. Legacy systems are vital elements of production in many organizations and are often the only complete and accurate source of business rules.

As software ages it begins to deteriorate as new functions are added and old functions are modified. Each software change makes the next change more difficult. Maintenance costs continue to increase until system operation is economically unsound or ceases. Eventually, the software must be replaced.

Legacy system replacement is difficult because of cost, time, and risk. Of these three factors, the risk of lost functionality in replacement systems is considered to be the most significant. As much as 90 percent of the replacement system functionality is likely to be the same as existing system functionality. A necessary first step in system replacement is an analysis of the existing system to ensure that functionality is included in the new system or intentionally eliminated.

Reverse engineering, the extraction of information at a level above program source code, is based on the need to understand software for maintenance and system replacement purposes. The difference in reverse engineering for maintenance and for replacement is primarily one of degree. Reverse engineering for maintenance requires precision; reverse engineering understanding for replacement requires more abstract understanding.

Programming languages consist of two components: syntax and semantics. Syntax specifies the way the elements of the language are used together to create valid statements. Semantics is the meaning associated with the syntactic structure. Programming language syntax is much simpler than natural language syntax. There is also a corresponding decrease in the semantic context. Because it has limited semantic content, legacy system source code is difficult to understand.

Program understanding (i.e., reading a program to extract its semantic content) is difficult because program structure, function, and purpose are not mutually exclusive nor collectively exhaustive. Program complexity exists in three forms: (a) logical - the number of possible paths through a program, (b) structural - the number of modules and their interrelationships, and (c) psychological - the characteristics of software that make it difficult for humans to understand (e.g., the number of IF statements, module size, and non-normal exits from decision statements).

A critical element in both forward and reverse engineering is semantic knowledge of the application domain. During the beginning phases of information systems development, a great deal of domain knowledge is necessary to describe functional requirements. As systems development progresses, this semantic domain knowledge is replaced by technical implementation knowledge. Unless domain knowledge is captured in life cycle documentation (e.g., functional descriptions, systems specifications, user manuals), it is lost during system implementation.

The limited semantic knowledge contained in programming languages does not allow retention of domain knowledge with source code except in the form of program comments. The universal reluctance of system developers to prepare documentation and programmers to prepare source code comments contributes to the problem of functional knowledge recovery.

System and program documentation is almost universally inadequate--it does not contain the semantic knowledge used to build a system, and more often describes the system as it was implemented in technical rather than functional terms. When source code is modified, the documentation is seldom updated even in cases where the original documentation contained domain knowledge. Within a relatively short period of time, system documentation and source code differ in existing system semantic content.

The problems associated with aging legacy systems began to be recognized in the late 1980s and early 1990s and became a driving force in reverse engineering research. Because of the massive amount of legacy system code in existence, research on reverse engineering focused on automated or computer-aided solutions. Artificial intelligence and knowledge-based reverse engineering techniques were extensively explored, and were found to be inadequate except for small programs with simple logic. Other reverse engineering techniques are more properly classified as reengineering techniques; they do not abstract knowledge about software systems at a level higher than program code.

A major conclusion resulting from a comprehensive review of the literature in the area is that reverse engineering, like original software development, is primarily a human activity; automatic reverse engineering of functional design information from program source code is an unsolvable problem for a computer. Source code is incomplete and must be augmented with the domain knowledge lost during systems development.

The domain analysis-based process reverse engineering methodology described in this research recognizes the need to replace missing domain knowledge. The top-down component of reverse engineering--the functional domain model produced with the assistance of knowledgeable users--serves to outline high-level functional key areas and tasks represented in source code. The bottom-up component of reverse engineering--the extraction of domain oriented program components--is guided by the structure of the domain model. In effect, the hierarchical structure of the domain model provides target slots into which low-level source code activities can be placed.

Program component extraction is accomplished in two steps. The first step consists of preparing a program structural model that identifies major inputs, outputs, and connectivity with other programs. Explanatory program header comments and in-line program paragraph notes are extracted and recorded in a program information database. Non-implementation dependent program paragraphs are extracted and stored in the program information database. The database is used to automatically prepare the program structure model. The program structure model reduces the need to work from source

code and is the first level of abstraction above the program level. The second step consists of using the program structure model to convert program paragraphs to their functional activity equivalents. This conversion is accomplished with the assistance of both application domain specialists and information system technicians. Functional activities and their narrative descriptions are then assigned to the appropriate structure within the top-down domain model. The completed domain model is verified with functional users and modified as required to present a functional description suitable for specifying a replacement system.

A prototype program information database was developed to support the methodology. The process reverse engineering methodology was evaluated against a test case made up of real programs. The results of the test case were extremely positive and demonstrated the approach feasibility. Two metrics suitable for predicting the amount of time required to reverse engineer a program were identified and evaluated.

Reverse engineering legacy system source code is, without question, a difficult task further complicated by poor documentation, programming, and maintenance practices. There is ample evidence to suggest that new systems being developed are not significantly better than those developed 10, 20, and 30 years ago; they are the legacy systems of tomorrow. There is a continuing and perhaps critical need for information technicians to learn and apply effective reverse engineering skills. Information system technicians may need to become specialists in reverse engineering--a field that will be identified as "software gerontology."

Appendix A

Glossary

Abstraction. A high-level representation made up of words and pictures at a level higher than a system and that accurately reveals the system, its components (data and function), and their interrelationships (Pfrenzinger, 1992).

Adiabatic. The volumetric compressibility/expandability of any aspect of the program information space with minimum loss of dependency information (Khan, 1994).

Application (data processing). All the functionalities used for a particular, identifiable, and discrete purpose (Grumann & Welch, 1992).

Architectural design (preliminary design, software product design). Identifies the software components, decoupling and decomposing them into processing modules and conceptual data structures, and specifying the interconnections among components (Fairley, 1985).

Call graph. A diagram that identifies the modules in a system or computer program and shows which modules call one another (ANSI/IEEE Std 610.12, 1990).

Computer program. A combination of computer instructions and data definitions that enable computer hardware to perform computational or control functions (ANSI/IEEE Std 610.12, 1990).

Concept phase. The period of time in the software development cycle during which the user needs are described and evaluated through documentation (i.e., statement of needs, feasibility study, system definition) (ANSI/IEEE Std 610.12, 1990).

Data flow diagram. A diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes (ANSI/IEEE Std 610.12, 1990).

Descriptiveness. The extent to which software contains information regarding its objectives, assumptions, inputs, processing, outputs, components, revision status, etc. (Peercy, 1981).

Design. Identifies software components (functions, data streams, and data stores) specifying relationships among components, specifying software structure, maintaining a record of design decisions, and providing a blueprint for the implementation phase (Fairley, 1985).

Design Recovery. A subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify "meaningful" higher-level abstractions beyond those obtainable directly by examining the system itself (Cross, Chikofsky & May, 1992).

Detailed design (software design specification). Concerned with how to package the processing modules and how to implement the processing algorithms, data structures, and interconnections among modules and data structures (Fairley, 1985).

Directed graph. A graph in which direction is implied in the internode connections (ANSI/IEEE Std 610.12, 1990).

Domain. A coherent set of systems that exhibits common features and functionality across existing and proposed instances. A domain may be defined as a vertical or horizontal component within a larger context, (e.g., window systems are a horizontal domain, microwave instrument firmware is a vertical domain) (Ogush, 1992).

Domain Analysis. The process of identifying and organizing knowledge about some class of problems--the problem domain--to support the description and solution of those problems (Arango & Prieto-Diaz, 1991).

Extraction (in reverse engineering). The process of extracting parts of a program, such as the extraction of the call tree or program slice; also involves the choice of particular modules to be examined or particular program paths (Howden & Pak, 1992).

Flow chart. A control flow diagram in which suitably annotated geometrical figures are used to represent operations, data, or equipment and arrows are used to indicate the sequential flow from one to another (ANSI/IEEE Std 610.12, 1990).

Forward engineering. The traditional process of moving from high-level representations and logical, implementation of a system; follows a sequence from the analysis of requirements through the design, and finally to an implementation (Cross, Chikofsky & May, 1992).

Functional decomposition. A type of modular decomposition in which a system is broken down into components that correspond to system functions and subfunctions (ANSI/IEEE Std 610.12, 1990).

Functional requirement. A requirement that specifies a function that a system or system component must be able to perform (ANSI/IEEE Std 610.12, 1990)..

Graph. A diagram or other representation consisting of a finite set of nodes and internode connections called edges or arcs (ANSI/IEEE Std 610.12, 1990).

Hierarchical decomposition. A type of modular decomposition in which a system is broken down into a hierarchy of components through a series of top down refinements (ANSI/IEEE Std 610.12, 1990).

Implementation. Translation of design specifications into source code, and debugging, documentation, and unit testing of source code (Fairley, 1985).

Job control language. A language used to identify a sequence of jobs, describe their requirements to an operating system, and control their execution (ANSI/IEEE Std 610.12, 1990).

Maintenance. The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment (ANSI/IEEE Std 610.12, 1990).

Module. A set of contiguous computer language statements which has a name by which it can be separately invoked (Peercy, 1981).

Morphogenic. Structure changing (Buckley, 1972).

Morphology. The study of structure or form employing a definite behavioral approach and methodology (Sage, 1977).

Morphostatic. Structure preserving (Buckely, 1972).

Narrow spectrum language. A language that covers only a limited range of abstractions; usually intended for a particular phase of development (Keller & Nance, 1993)

Partitioning. Dividing or disaggregating an issue into parts such that it can be more effectively represented or more easily understood through a description of the parts (Sage, 1977).

Program. A description of a method of computation that is expressible in a formal language (Partsch & Steinbrüggen, 1983).

Program plan. An abstract representation of an algorithmic structure; it identifies the building components of an algorithm in terms of a set of atomic program elements; it also identifies the proper arrangement of components (Harandi & Ning, 1988).

Preliminary design. The process of analyzing design alternatives and defining the architecture, components, interfaces and timing and sizing estimates for a system or component (ANSI/IEEE Std 610.12, 1990).

Program scheme. The representation of a class of related programs; originates from a program by parameterization. Programs can be obtained from program schemes by instantiating the scheme parameters (Partsch & Steinbrüggen, 1983).

Program slicing. The process of stripping a program of statements without influence on a given variable at a given statement; slices are generally not contiguous pieces, but contain statements scattered throughout code (Weiser, 1982).

Recapture (technologies). The attempt to recover the original design in an existing software system by using reverse engineering and various program-understanding tools (Müller, Tilley, Orgun, Corrie, & Madhavji, 1992).

Redesign. Improving an existing system by examining the functionality and making enhancements and modifications without regard to the existing code (Ochs, 1993).

Redevelopment. Using an essential view of an information system to construct an improved system (Ochs, 1993).

Redocumentation. The production of a semantically equivalent representation (often paper based) of the target system at whatever level of abstraction is being addressed (Frazer, 1992).

Reengineering. Software engineering activities designed to effect the transformation of existing systems in order to achieve conformity with prevailing programming standards, to implement in high-order languages for easier maintenance, to rehost to other hardware platforms, or to retarget to other computer system architectures; usually initiated to transform existing "bad" systems to new "good" systems (Yu, 1991).

Requirements analysis. The process that identifies the basic functions of the software component in a hardware/software/people system; emphasis is on what the software is to do and the constraints under which it will perform its function (Fairley, 1985).

Requirements specification. A document that specifies the requirements for a system or component. Typically included are functional requirements, performance requirements, interface requirements, design requirements and development standards (ANSI/IEEE Std 610.12, 1990).

Restructuring. The transformation of a software system from one representation to another, usually at the same relative abstraction level, while preserving the subject system's external behavior (i.e., functionality and semantics) (Cross, Chikofsky & May, 1992).

Reuse. Software engineering activities which focus on the identification of reusable software for straight import, reconfiguration, and adaptation for new computing system applications (Yu, 1991).

Reverse engineering. **1.** Taking existing programs and their associated file and database descriptions and raising their design objects from the implementation ("how") level to the specification ("what") level of design (Bachman, 1988). **2.** The process of analyzing a subject system in order to identify the system's components and their interrelationships and to create representations of the system, possibly at a higher level of abstraction (Cross, Chikofsky, & May, 1992). **3.** The process of gaining a

basic understanding of a legacy system; the objective is to identify all components of the system and understand what the system does in business terms (Connal & Burns, 1993). **4.** The process of taking existing applications (database and programs) and recycling them into a format that can be forward engineered (Kerr & McGovern, 1991). **5.** The process of transforming or moving from one level of description of a system to a level which is regarded as more abstract or "earlier" in terms of the standard life cycle (Lano & Haughton, 1994). **6.** A process that uses existing code to extract and document a higher level model of the as-built information system (Ochs, 1993). **7.** Software activities pertaining to computer-aided extraction of specifications, design, and software components from existing software systems; implies derivation of abstract specifications from existing "good"" software systems and usually includes transverse engineering steps (Yu, 1991).

Semantics. The relationships of symbols or groups of symbols to their meaning (ANSI/IEEE Std 610.12, 1990).

Software. The programs and documentation which result from a software development process (Peercy, 1981).

Software development process. The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes installing and checking out the software for operational use (ANSI/IEEE Std 610.12, 1990).

Software documentation. The set of requirements, design specifications, guidelines, operational procedures, test information, problem reports, etc., which in total form the written description of the programs(s) from a software development process (Peercy, 1981).

Software engineering. The application of scientific principles to: (1) the orderly transformation of a problem into a working software solution, and (2) the subsequent maintenance of that software through the end of its useful life (Davis, 1988).

Software psychology. The study of human performance in using computer and information systems; its goal is to facilitate the human use of computers (Shneiderman, 1980).

Software system. All the elements, such as the source code, the JCL for constructing and running the system, databases, object code, documentation, design information, requirements and specification details; it is also the knowledge and expertise of the analysts and programmers who developed the system plus the knowledge and expertise of the maintenance programmers who are carrying out the various maintenance tasks in the continuing evolution of the system (Munro, 1992).

Source code. Computer instructions and data definitions expressed in a form suitable for input into an assembler, compiler, or other translator (ANSI/IEEE Std 610.12, 1990).

Source program. A computer program that must be compiled, assembled, or otherwise translated in order to be executed by a computer (ANSI/IEEE Std 610.12, 1990)..

Specification. A recorded document of any software life-cycle activity (Cross, Chikofsky & May, 1992).

Statement. In a programming language, a meaningful expression that defines data, specifies program actions, or directs the assembler or compiler (ANSI/IEEE Std 610.12, 1990).

Structure. A hierarchy of information sets in which the elements at each level are related (ordered) in terms of either sequence, alteration, repetition, concurrency or recursion (Orr, 1981).

Structure chart. A diagram that identifies modules, activities or other entities in a system or computer program and shows how larger or more general entities break down into smaller, more specific entities (ANSI/IEEE Std 610.12, 1990)..

Structural abstraction. The process of making simplifying reductions in program structures; it can be described in terms of sequences, branching substructures, and loops. It can also be described in terms of basic program structures: sequencing, conditional branching, and iteration (Howden & Pak, 1992).

Syntax. The structural or grammatical rules that define how the symbols in a language are to be combined to form words, phrases, expressions, and other allowable constructs (ANSI/IEEE Std 610.12, 1990).

System. A collection of components organized to accomplish a specific function or set of functions (ANSI/IEEE Std 610.12, 1990).

System life cycle. The period of time that begins when a system is conceived and ends when he system is no longer available for use (ANSI/IEEE Std 610.12, 1990).

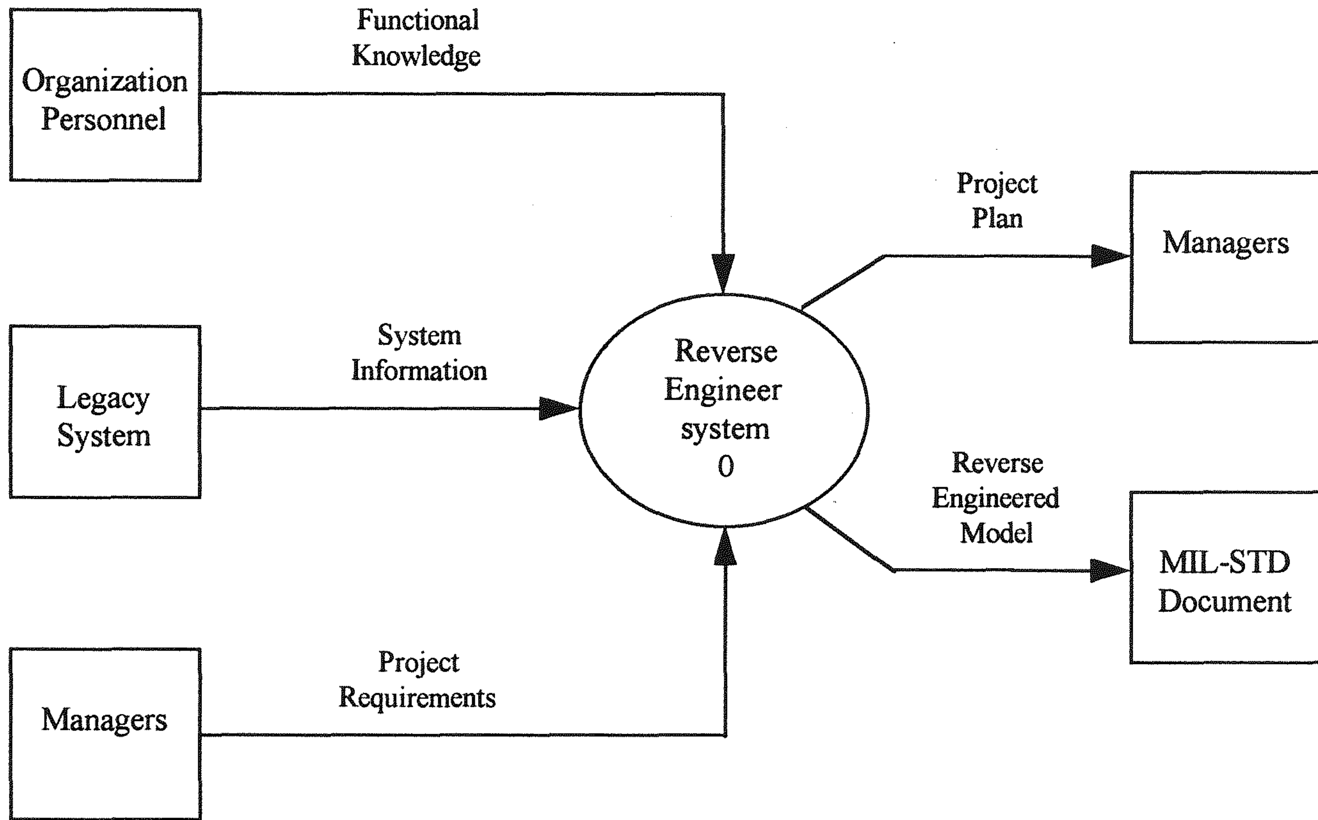
Teleology. The study of the purpose of things; considers a system to be organized as a set of elements directed towards the realization of goals (Karakostas, 1990).

Understandability. The extent to which the purpose and organization of software are clear to the reviewer (Peercy, 1981).

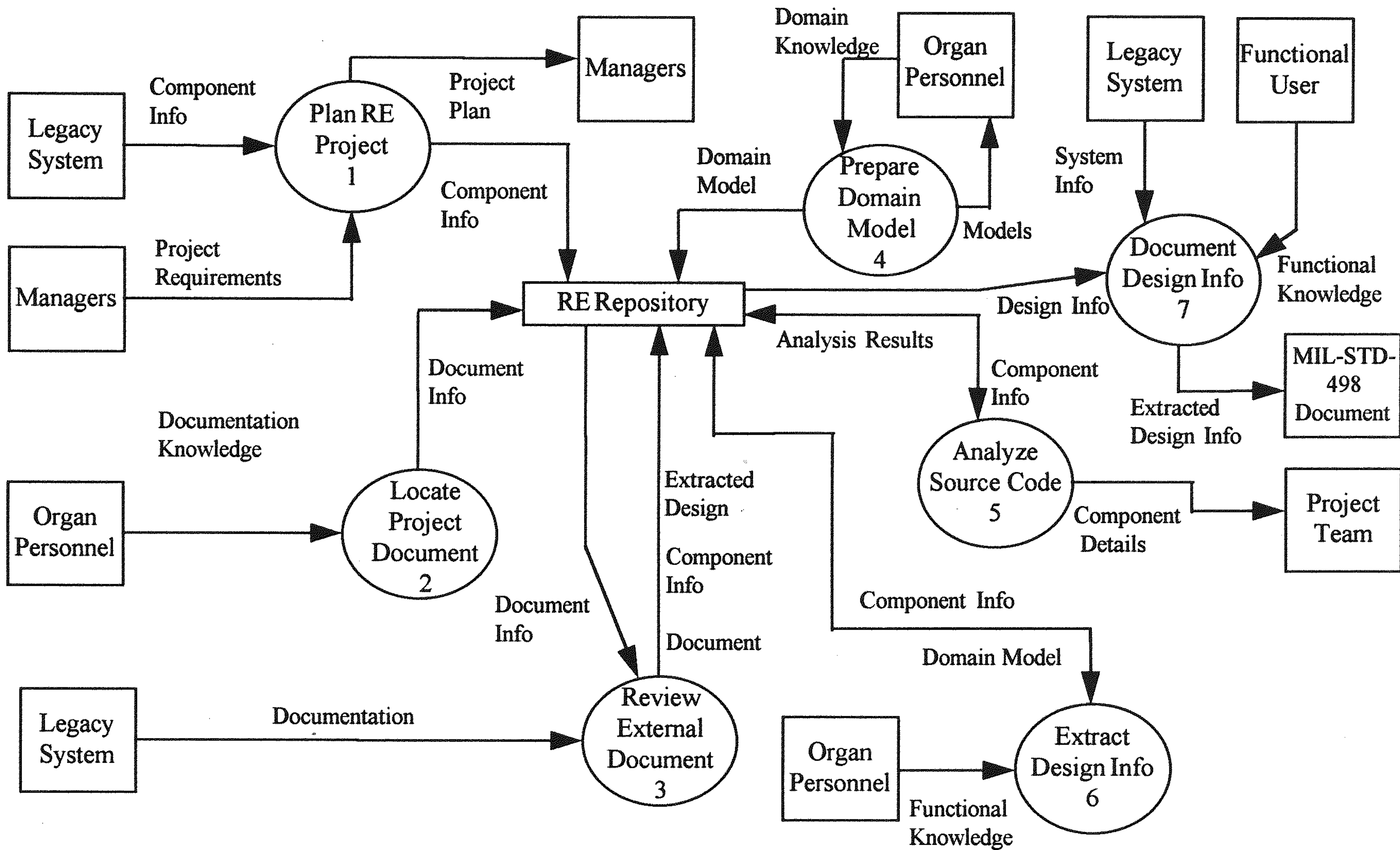
Wide spectrum language. A language in which all levels of abstraction from system requirements to programs are expressible (Keller & Nance, 1993).

Appendix B

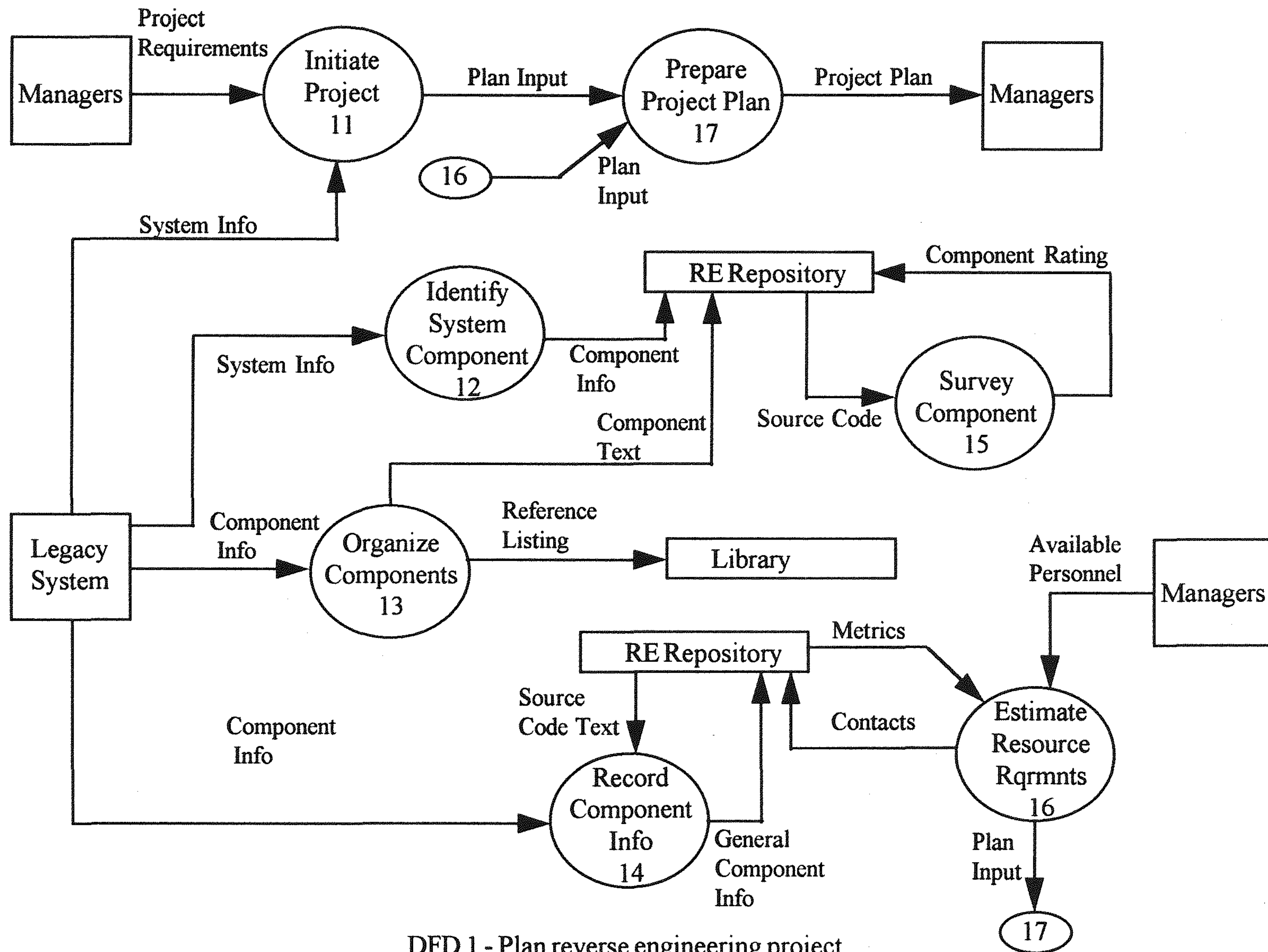
Reverse Engineering Methodology
Conceptual Model
Data Flow Diagrams
and
Process Descriptions



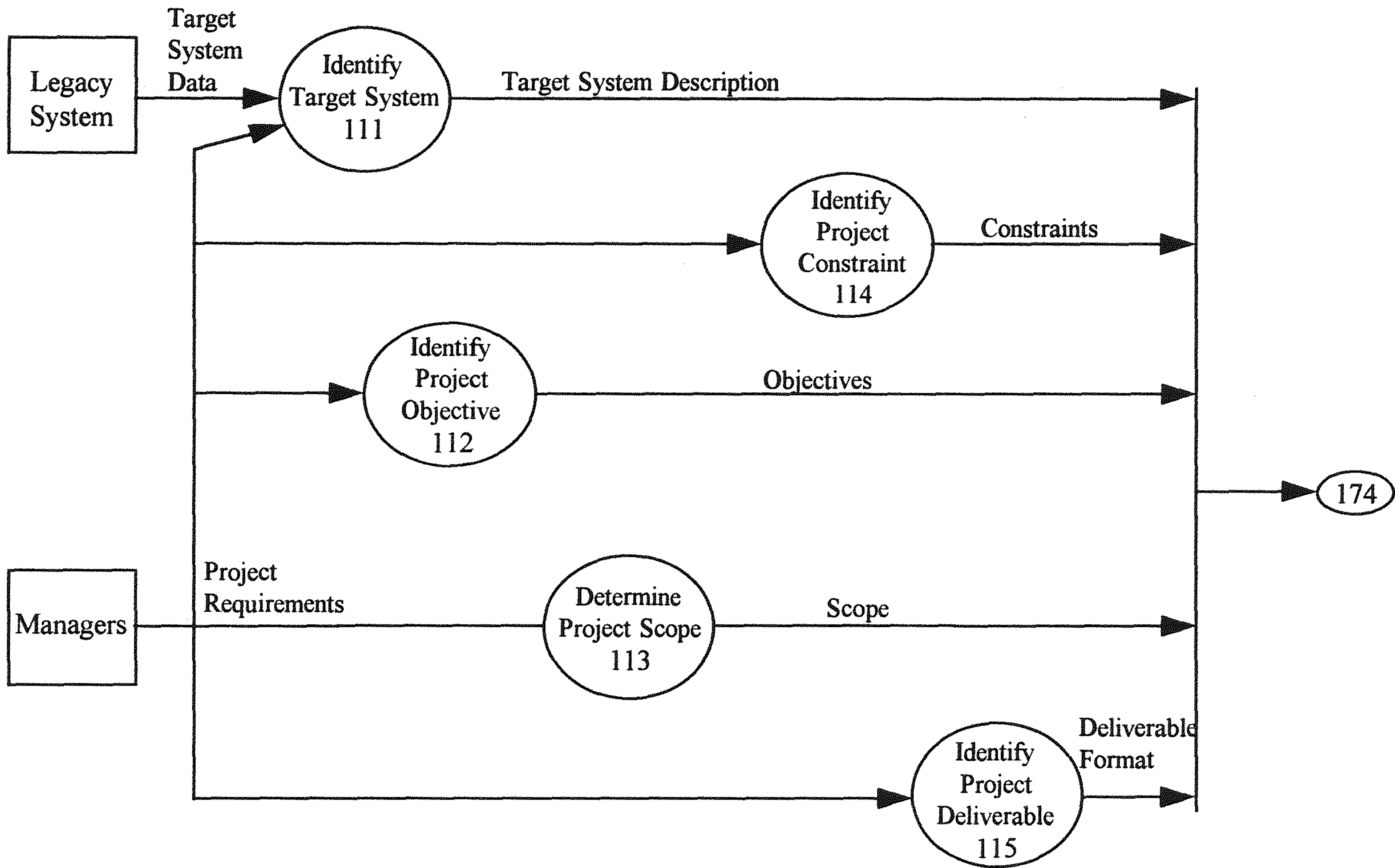
DFD Context diagram



Level 0 Diagram



DFD 1 - Plan reverse engineering project



DFD 1.1 - Initiate project

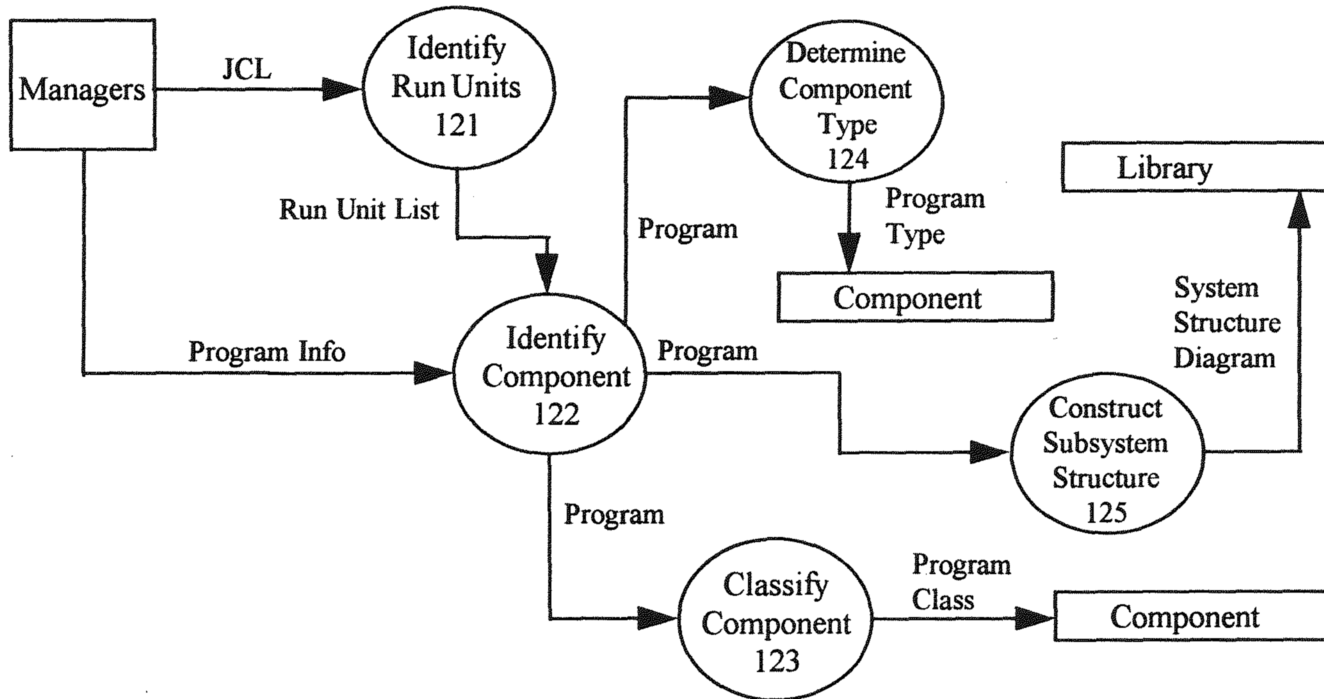
1.1.1 Identify Target System. The system to be reversed engineered is clearly identified using the project requirements directive and available legacy system information.

1.1.2 Identify Project Objective. The reverse engineering project objectives are identified from the project requirements directive. The objective is clearly stated to ensure the project team understands why the project is being undertaken and what the expectations are.

1.1.3 Determine Project Scope. The scope of the project is determined from the project requirements directive and is clearly specified to ensure the project team understands the boundaries.

1.1.4 Identify Project Constraint. Constraints, such as time allotted for the project, required completion data, budget, number of personnel to be assigned, and similar limitations are identified from the project requirements directive.

1.1.5 Identify Project Deliverable. The format and content of the final project deliverable is identified from the project requirements directive. The deliverable requirement is described in sufficient detail to allow the project team to prepare the final document.



DFD 1.2 - Identify system components

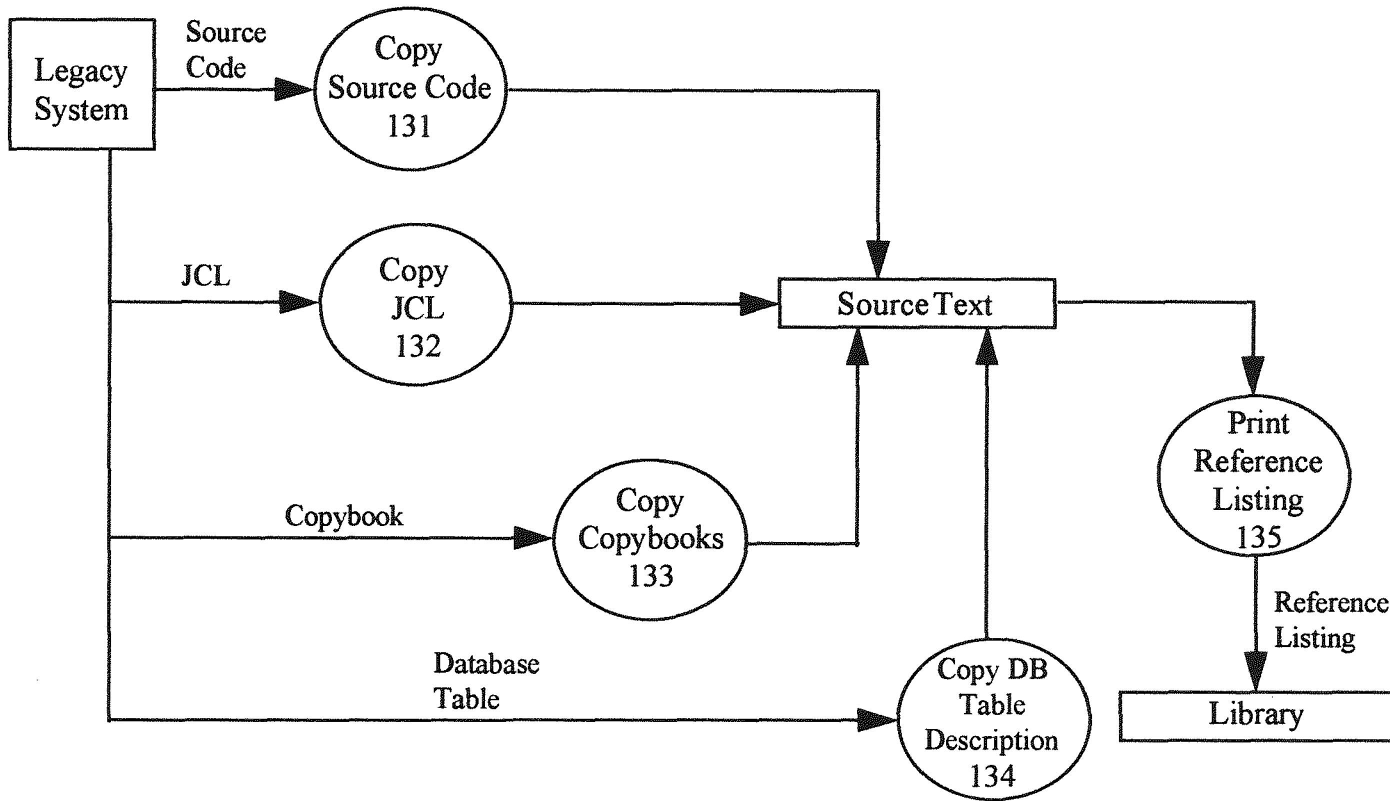
1.2.1 Identify Run Unit. Individual batch system run units (jobs) for the specified system are identified from available operations documents and program libraries.

1.2.2 Identify Component. Individual components (programs) of the target system jobs are identified from available documentation or with the assistance of operations personnel. On-line programs are identified by extracting program identifiers from CICS tables and program libraries.

1.2.3 Classify Component. Components are classified as programs or subprograms according to their system use.

1.2.4 Determine Component Type. Component types (e.g., batch, on-line) are determined by examining the general structure of a program or by its location in a program library.

1.2.5 Create Subsystem Structure. A preliminary model of the hierarchical program structure is created by diagramming the execution sequence of batch programs and the calling structure of on-line programs.



DFD 1.3 - Organize components

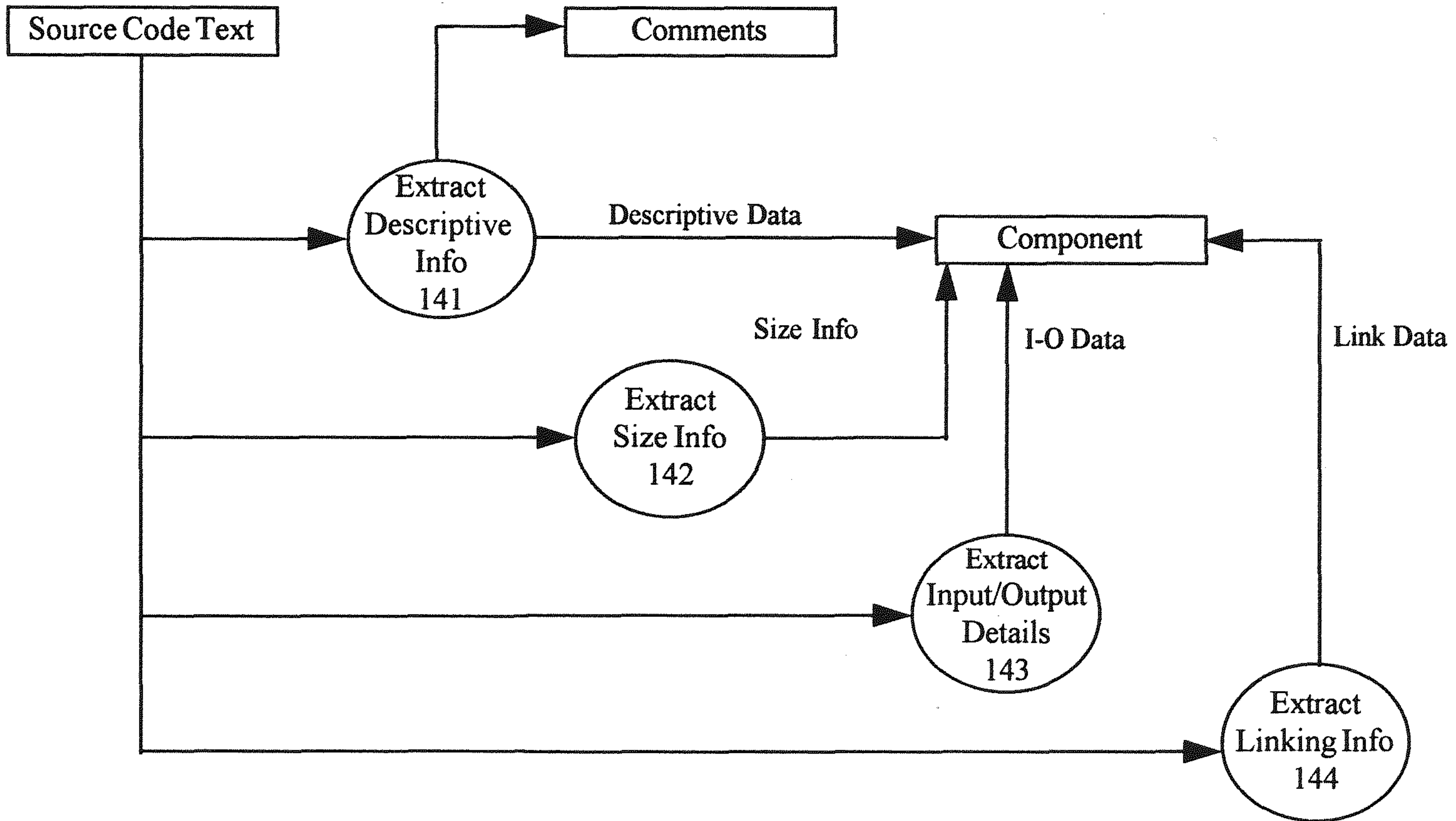
1.3.1 Copy Source Code. System components are copied from the mainframe system in .TXT format as individual files and stored on electronic media.

1.3.2 Copy Job Control Language (JCL). The JCL for the target system is copied from the mainframe system in .TXT format as individual files and stored on electronic media.

1.3.3 Copy Copybooks. Copybooks containing standard record layouts and other commonly used data structures are copied from the mainframe system in .TXT format as individual files and stored on electronic media.

1.3.4 Copy Database Table Descriptions. Database table descriptions (data structure layouts) are copied from the mainframe system in .TXT format and stored on electronic media.

1.3.5 Print Reference Listing. Documentation in .TXT files are converted to personal computer word processing files and formatted for printing. Small font size and two-column printing is used to reduce the volume of printed material. Individual files are retained in the word processing format for later review.



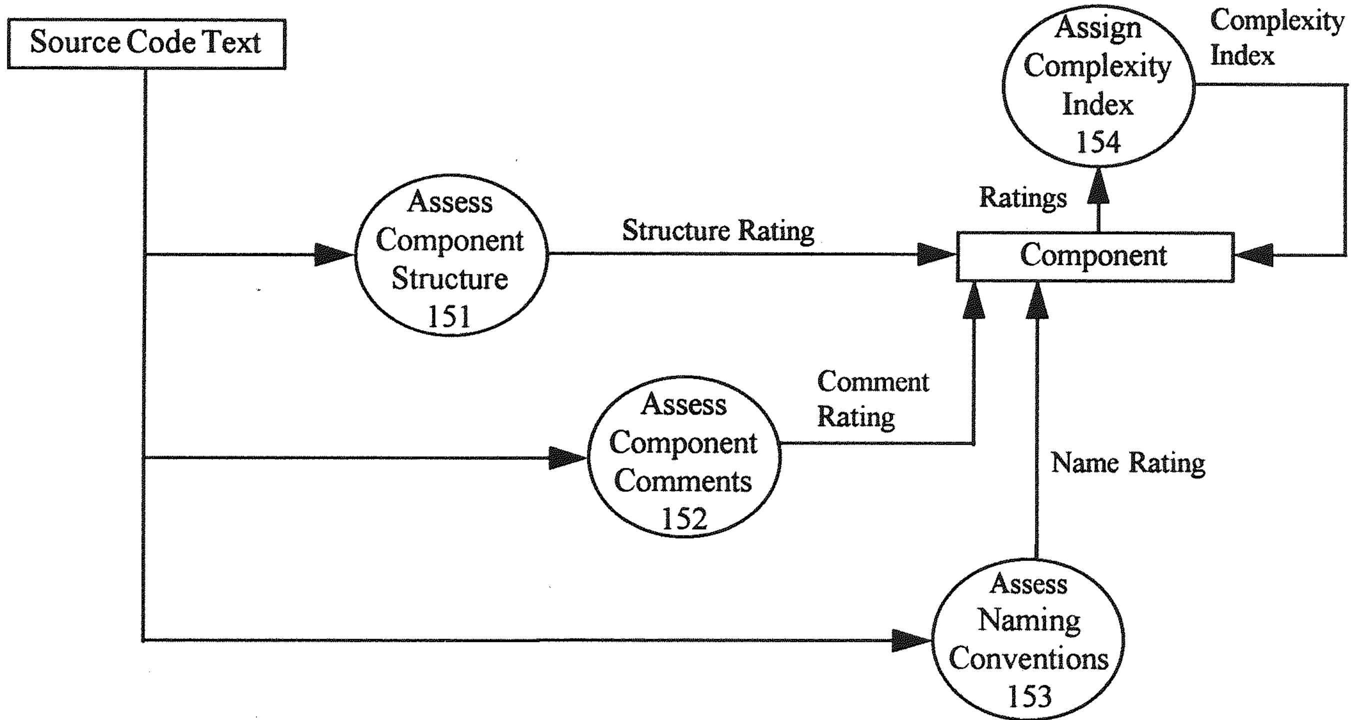
DFD 1.4 - Record component information

1.4.1 Extract Descriptive Information. Basic descriptive data for a system component is extracted from its reference listing and recorded in the reverse engineering (RE) repository. Header comments, date written, number of modifications, number of authors, and similar information is recorded.

1.4.2 Extract Size Information. Size information is extracted from a component reference listing and recorded in the RE repository. Size information includes information such as source lines of code, data division lines of code, and procedure division lines of code.

1.4.3 Extract Input and Output Details. Input and output details are extracted from a component reference listing and recorded in the RE repository. Information collected includes the number of files accessed, number of reports generated, number of screens associated with the component, and number of accessed database tables.

1.4.4 Extract Linking Information. The number of links (calling/called relationships) is extracted from a component reference listing and recorded in the RE repository.



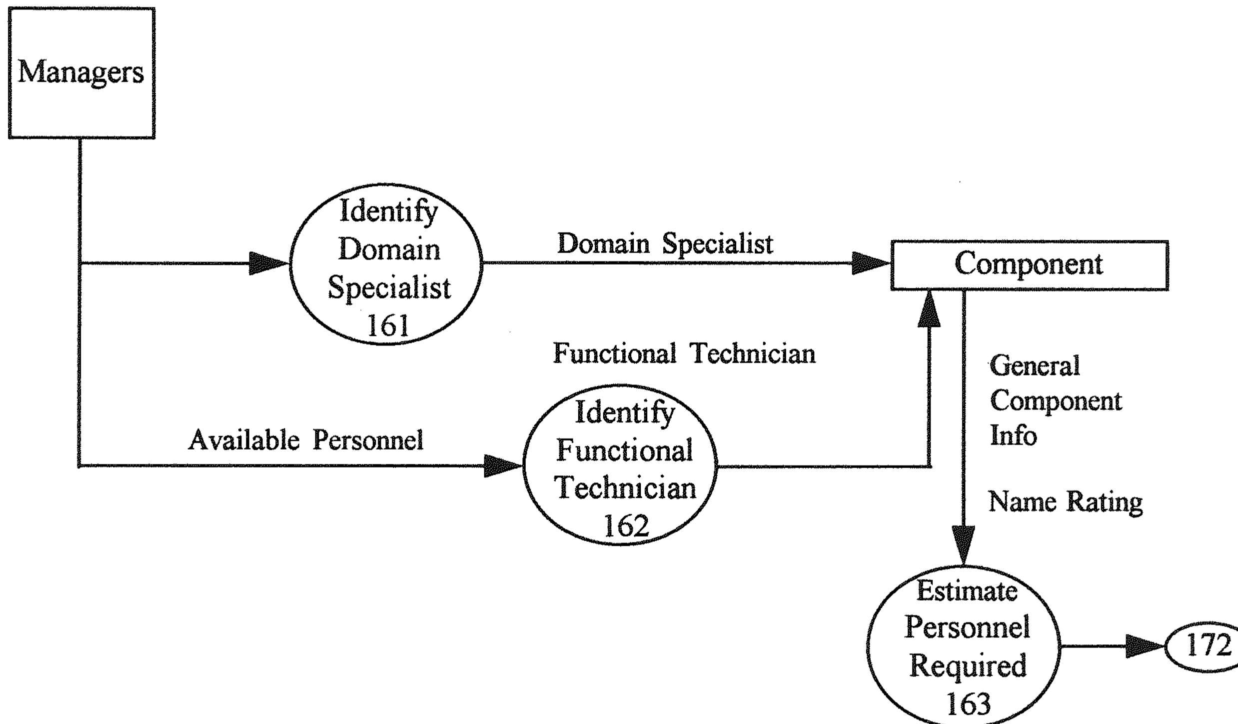
DFD 1.5 - Survey components

1.5.1 Assess Component Structure. A component reference listing is used to assess the degree of program structure. The number of GO TO statements used, the number of PERFORM statements used, and the total number of COBOL paragraphs in the component are used to arrive at a rating stored in the RE repository.

1.5.2 Assess Component Comments. A reference listing is used to count the number of comment lines in a component's source code. The number of in-line comments and the information content of the comments are used to arrive at a rating which is stored in the RE repository.

1.5.3 Assess Naming Conventions. A reference listing is used to assess the uniformity and clarity of both variable names and paragraph names in a component's source listing. COBOL names may be 30 characters long. The average number of characters in variable and COBOL paragraph names and an assessment of the name meanings is used to assign a rating stored in the RE repository.

1.5.4 Assign Complexity Index. Structure, comment, and naming ratings are retrieved from the RE repository and used with other component details to calculate a preliminary complexity index for each target system component. The complexity index is stored in the RE repository.

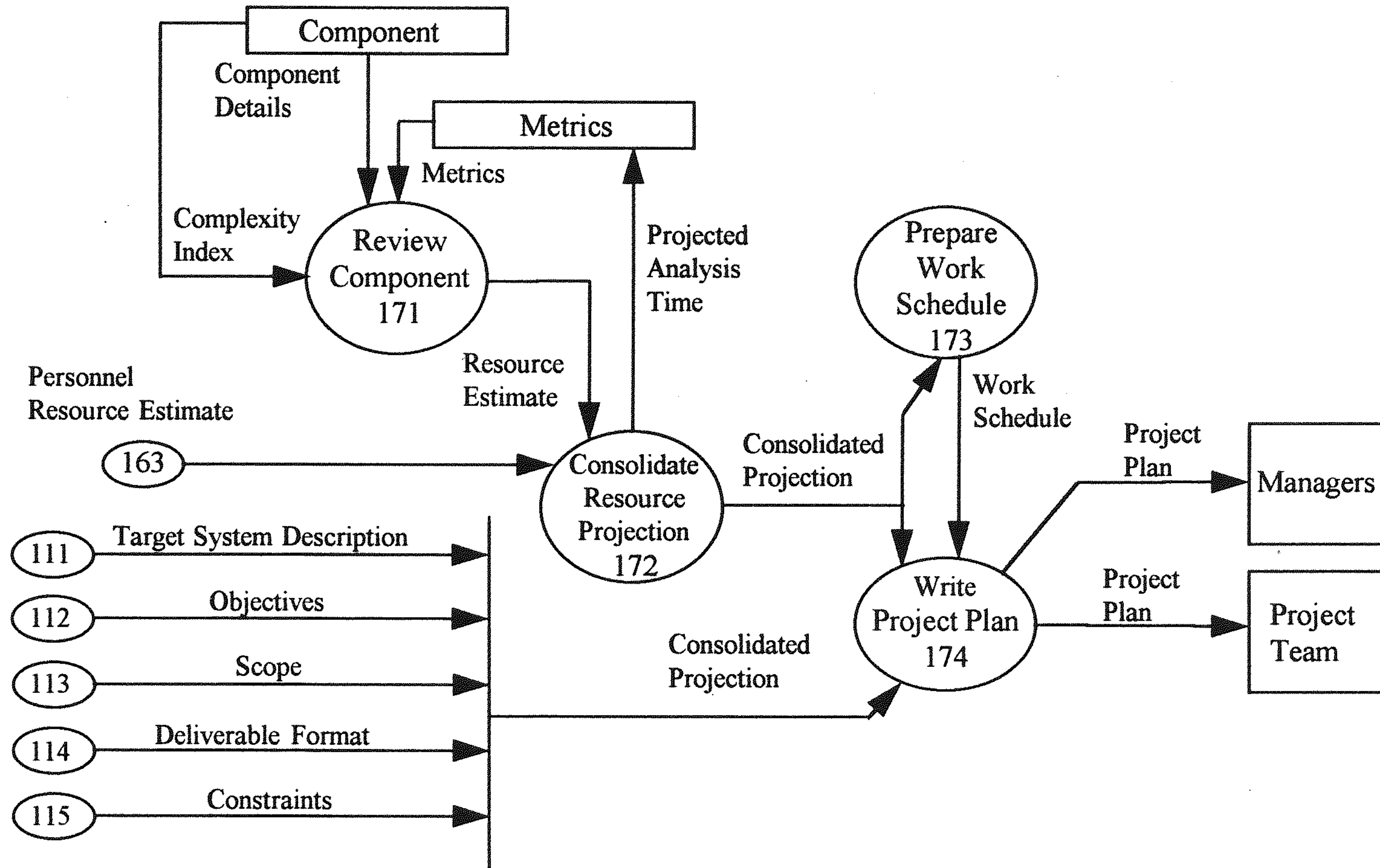


DFD 1.6 - Estimate organization resource requirements

1.6.1 Identify Domain Specialist. Managers and functional users are interviewed to identify organizational personnel capable of providing expert domain knowledge to the reverse engineering team. Depending on the complexity of the target system, several specialists may be identified for each major domain area.

1.6.2 Identify Functional Technician. Managers and functional users are interviewed to identify technical personnel with the greatest functional and technical knowledge of the target system. Maintenance programmers are good candidates for providing technical support to the reverse engineering team.

1.6.3 Estimate Personnel Required. The amount of time required of organizational personnel is estimated based on the complexity of the target system components and the application domain. Time requirements are based on creating the application domain model and periodic meetings with reverse engineers to discuss functional aspects of source code.



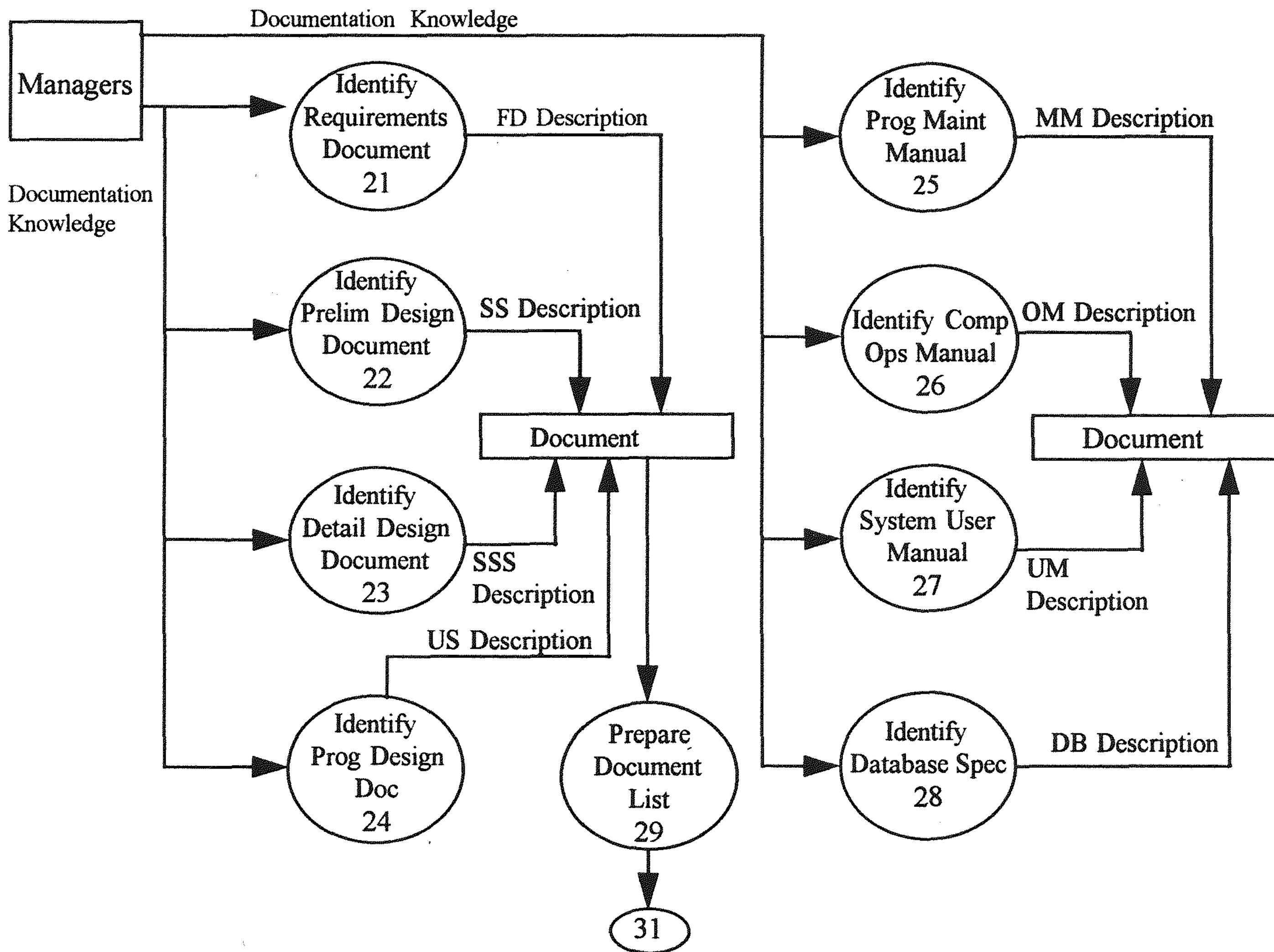
DFD 1.7 - Prepare project plan

1.7.1 Review Component. Target system component details are reviewed to establish data upon which to base resource estimates. Preliminary survey complexity indexes, component size information, and metrics derived from previous reverse engineering projects are used to project the resources required to reverse engineer a system component.

1.7.2 Consolidate Resource Projection. Projected resource requirements for individual components are consolidated to form an overall projection of the required reverse engineering effort.

1.7.3 Prepare Work Schedule. The consolidated resource projection is combined with personnel resource estimates to prepare a work schedule to complete the reverse engineering project.

1.7.4 Write Project Plan. The target system description, objectives, scope, constraints, deliverable format, and work schedule are used to write the project plan. The plan is delivered to organizational management and provided to the reverse engineering team.



DFD 2 - Locate project documentation

2.1 Identify Requirements Documentation. Organizational personnel are interviewed to determine the existence and location of target system requirements documentation. In the military environment, administrative system documentation requirements are specified in Department of Defense Standards (before 1995 DOD-STD-7935 or DOD-STD-7935A, the predecessor to MIL-STD-498).

2.2 Identify Preliminary Design Document. Organizational personnel are interviewed to determine the existence and location of a preliminary design document. Legacy system design specifications are typically recorded in a system specification (SS). The SS is normally a life cycle document.

2.3 Identify Detailed Design Document. Organizational personnel are interviewed to determine the existence and location of a detailed design document. In a simple system, all design specifications may be contained in an SS. In more complex systems, each subsystem design is documented in a separate subsystem specification (SSS). The SSS is usually a life cycle document.

2.4 Identify Program Design Document. Organizational personnel are interviewed to determine the existence and location of a program design specification. Depending on the age of the legacy system, this information may be found in a program specification (PS) or a software unit specification (US). Individual program specifications may be presented in separate documents or in separate sections of a single document. The PS and the US are not normally life cycle documents.

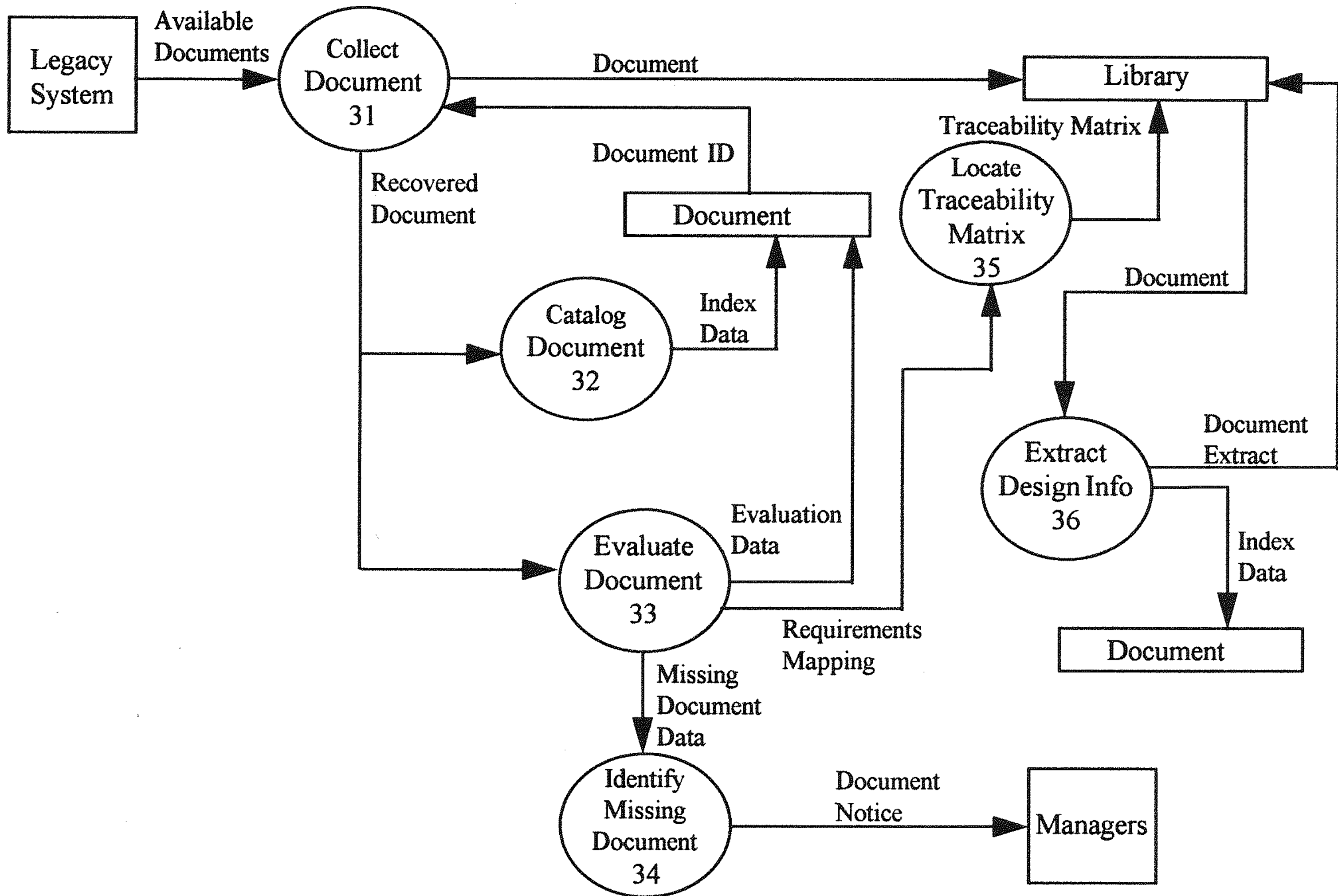
2.5 Identify Program Maintenance Manual. Organizational personnel are interviewed to determine the existence and location of a program maintenance manual (MM). The MM, a life cycle document, is used to support ongoing system maintenance.

2.6 Identify Computer Operations Manual. Organizational personnel are interviewed to determine the existence and location of the computer operations manual (OM). The OM is normally a life cycle document. The OM describes the individual jobs of a batch system.

2.7 Identify System User Manual. Organizational personnel are interviewed to determine the existence and location of a system user's manual (UM). The UM is normally a life cycle document.

2.8 Identify Database Specification. Organizational personnel are interviewed to determine the existence and location of a database specification (DB). The DB is normally a life cycle document.

2.9 Prepare Document List. A document list identifying the name and location of system reference material is prepared to support document collection.



DFD 3 - Review external documentation

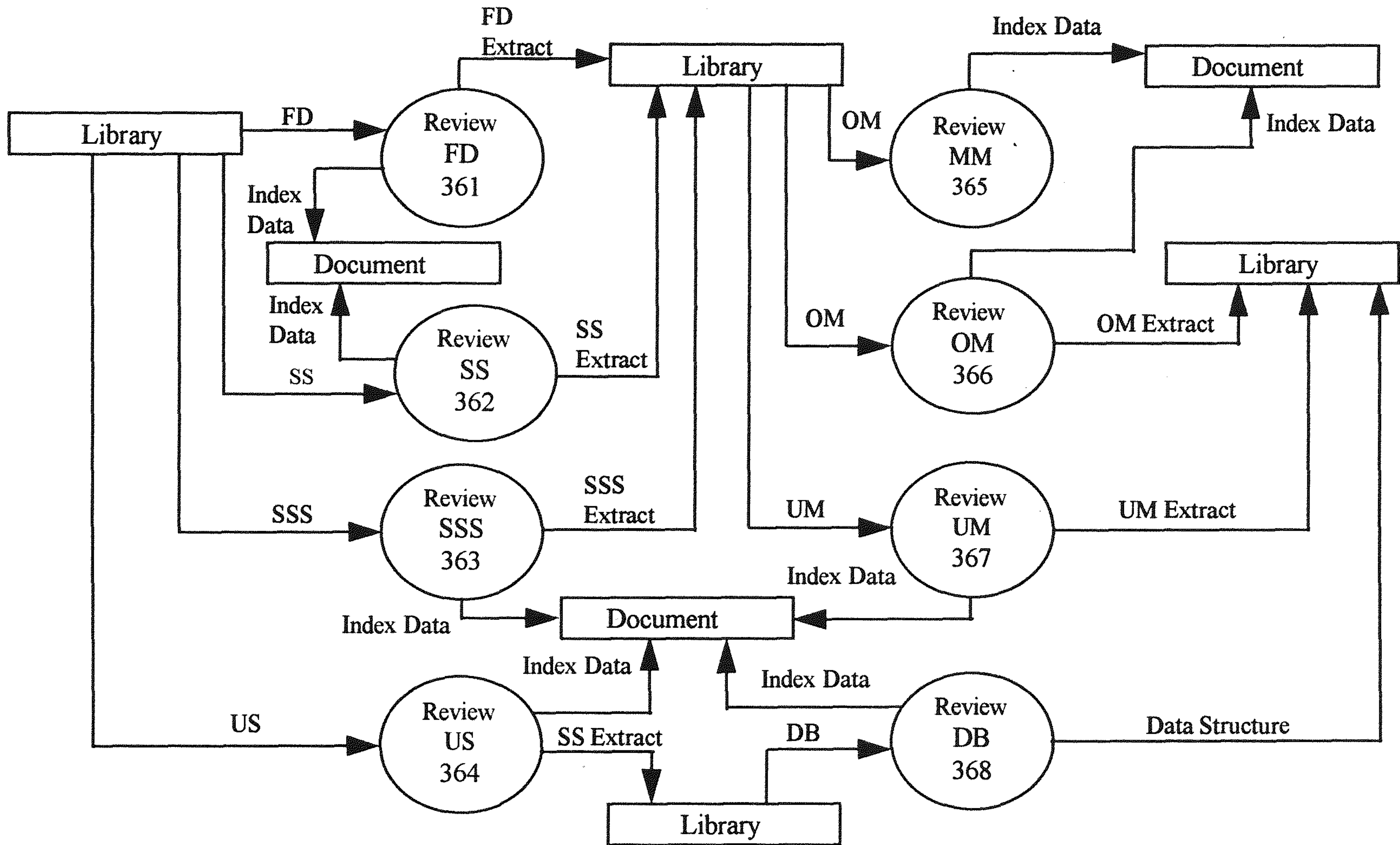
3.1 Collect Document. Available target system documentation is collected or copied and placed in the RE library.

3.2 Catalog Document. Individual documents are described and indexed in the RE repository.

3.3 Evaluate Document. Individual documents are evaluated for currency, correctness, completeness, and potential value in supporting the reverse engineering effort.

3.4 Identify Missing Document. Documents not found during the collection process or documents inadequate for reverse engineering are identified. Management is notified of the deficiencies. Additional interviews with organizational personnel may be scheduled to identify informal documentation suitable for replacing missing or inadequate documents.

3.5 Locate Traceability Matrix. Many military systems include a requirements traceability matrix that maps functional requirements from the FD to the SS/SSS to the PS. If such a document is located, it is copied and placed in the RE library.



DFD 3.6 - Review document

3.6.1 Review Functional Description (FD). The FD is reviewed to identify useful sections for the reverse engineering effort. Portions of Section 2 (Systems Summary - Proposed Methods and Procedures), Section 3 (Detailed Characteristics - Functional Area System Functions), and Section 4 (Design Considerations - System Functions) are extracted for the RE library. Index data is added to document files.

3.6.2 Review Systems Specification (SS). The SS is reviewed to identify useful sections for the reverse engineering effort. Portions of Section 2 (Summary of Requirements - System Functions) and Section 4 (Design Details - System Logical Flow) are extracted for the RE library. Index data is added to document files.

3.6.3 Review Subsystem Specification (SSS). If included in the system documentation, the SSS is reviewed to identify useful sections for the reverse engineering effort. Portions of Section 2 (Summary of Requirements - System Functions) and Section 4 (Design Details - System Logical Flow/System Data/Software Unit Descriptions) are extracted for the RE library. Index data is added to document files.

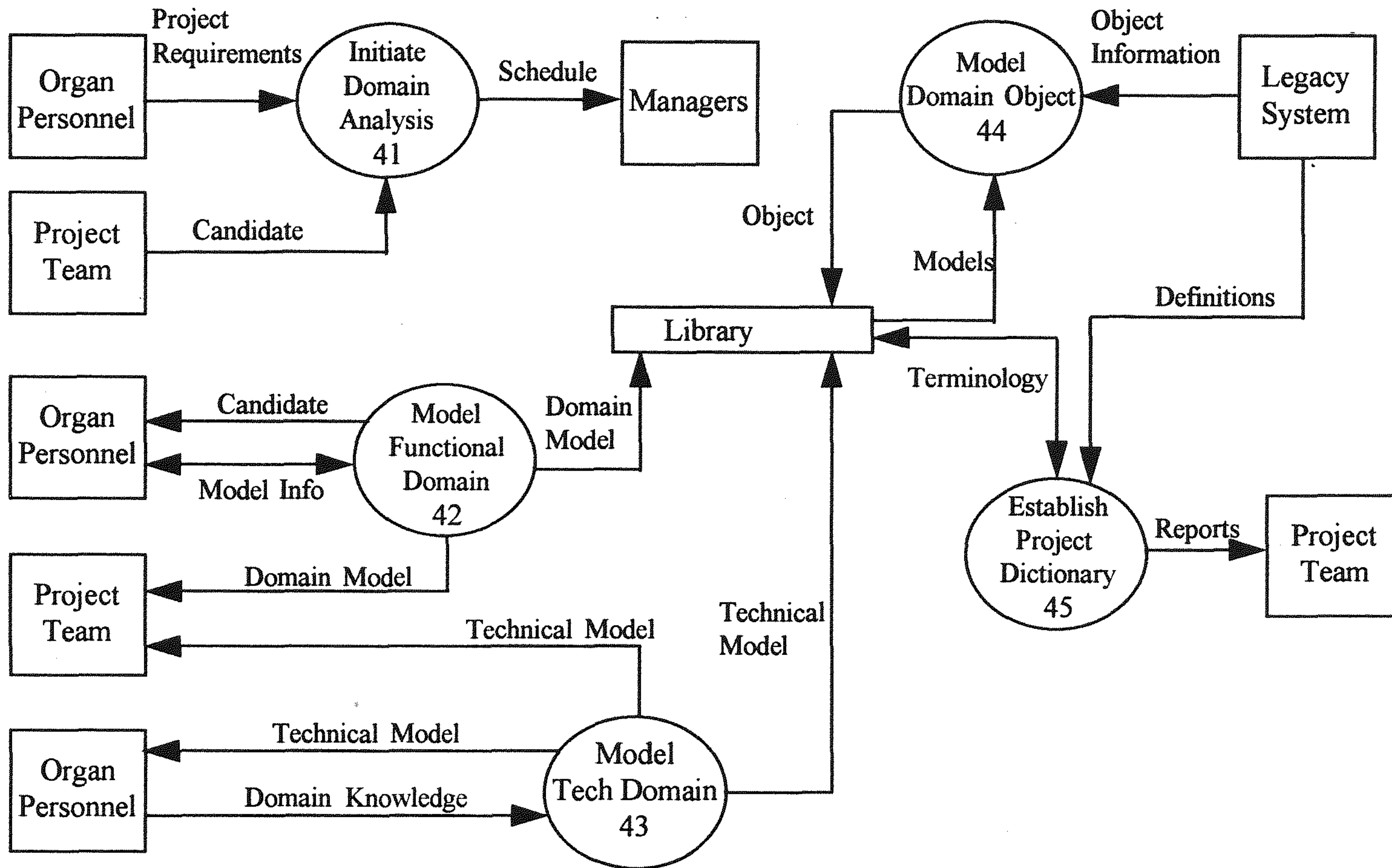
3.6.4 Review Software Unit Specification. Application software may be specified in program specifications (PS) or software unit specifications (US). The PS or US, if available, is reviewed for suitability. Extracts of Sections 2 (Summary of Requirements - Software Unit Description/Software Unit Functions) and Section 3 (Environment - Interfaces) are added to the RE library. Index data is added to document files.

3.6.5 Review Program Maintenance Manual (MM). The MM is reviewed to recover high-level structure information. Portions of Section 2 (System Description - System Organization/System Requirements Cross Reference) and Section 5 (Software Unit Maintenance Procedures) are extracted and placed in the RE library. Index data is added to document files.

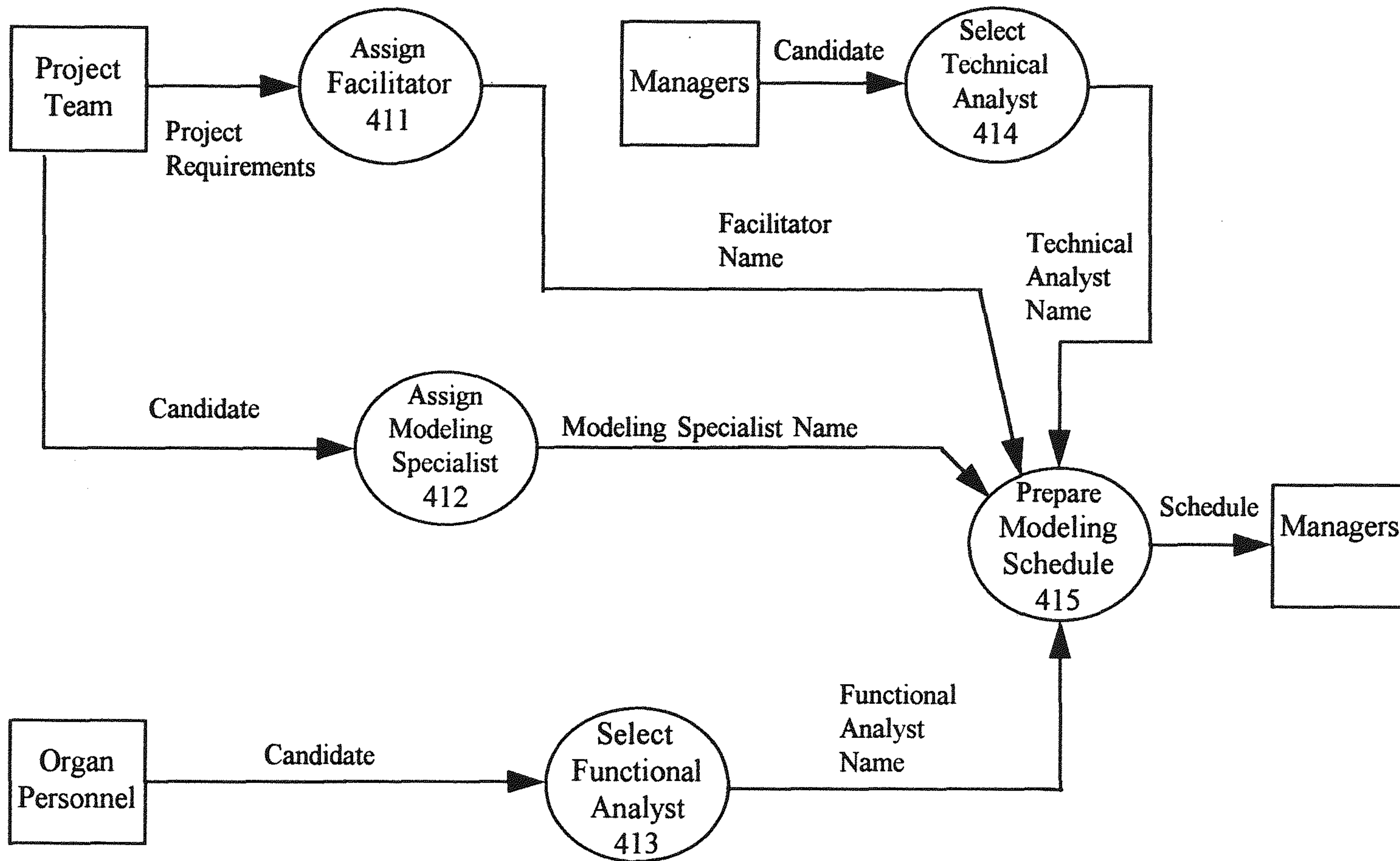
3.6.6 Review Computer Operations Manual (OM). The OM is reviewed to identify useful material. Portions of Section 2 (System Overview - System Organization/Software Inventory/Report Inventory/Processing Overview) and Section 3 (Description of Runs - Run Inventory/Run Description) are extracted for the RE library. Index data is added to document files.

3.6.7 Review User Manual (UM). The UM is reviewed for material of potential use. Most of the material contained in the UM is found in other document types in different formats, but general information contained in Section 4 (Processing Reference Guide) may be extracted and placed in the RE library. Index data is added to document files.

3.6.8 Review Database Specification (DB). The DB is reviewed to extract data structure information for *data* reverse engineering. Conceptual, logical, and physical data models may be documented in the DB. For older legacy systems, this document should describe the various master files supporting the system.



DFD 4 - Analyze application domain



DFD 4.1 - Initiate domain analysis

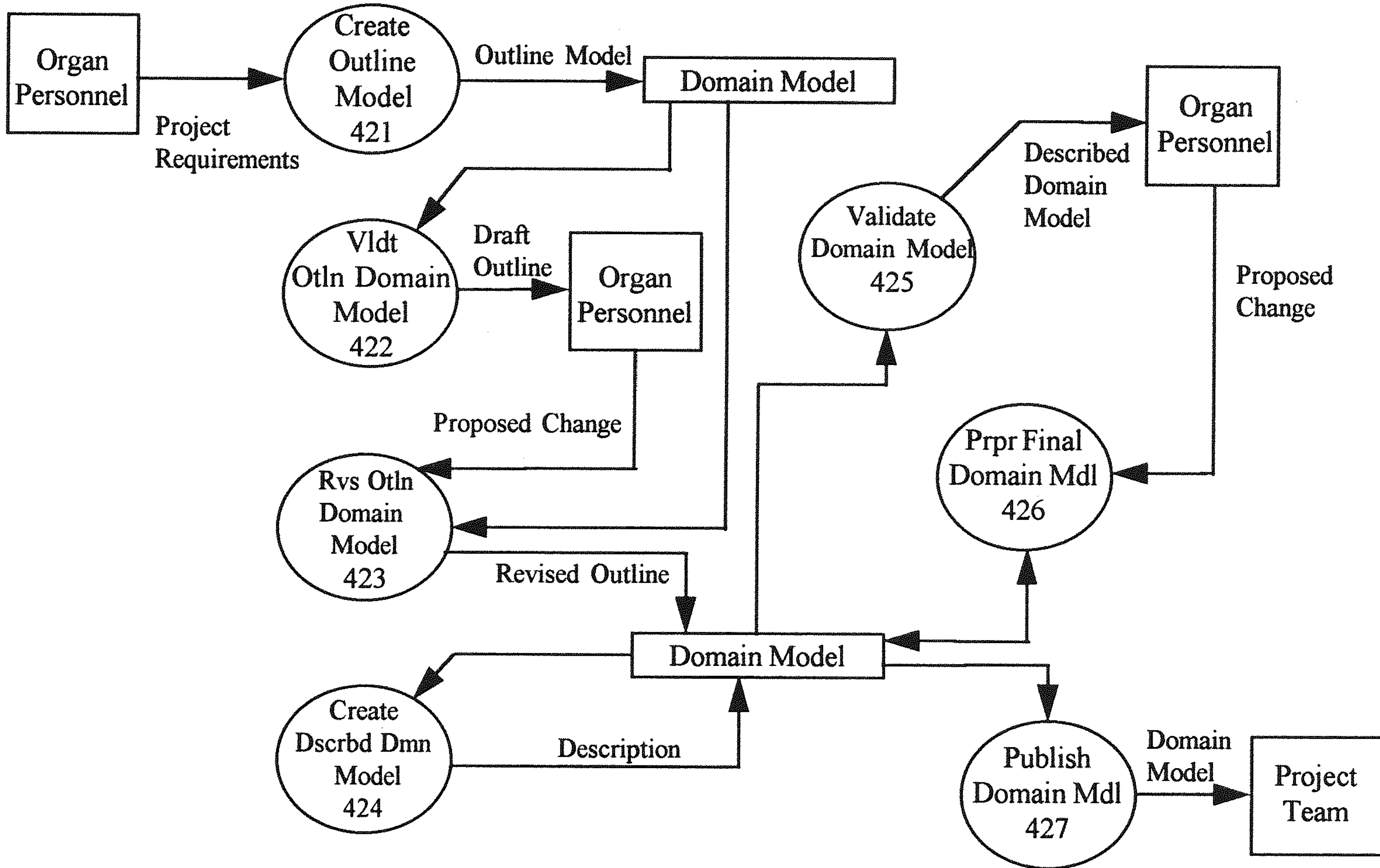
4.1.1 Assign Facilitator. A facilitator with experience in process and data modeling is assigned to the reverse engineering project. The ideal facilitator has previous knowledge and experience in the application domain; however, elicitation and modeling skills are more important than domain knowledge.

4.1.2 Assign Modeling Specialist. Modeling specialists with extensive experience in functional process modeling and conceptual data modeling are assigned to the reverse engineering project. After completion of the domain modeling sessions, the modeling specialists will be used as lead reverse engineers. The domain knowledge gained during the modeling sessions allows the modelers to begin the reverse engineering with some knowledge of the application area.

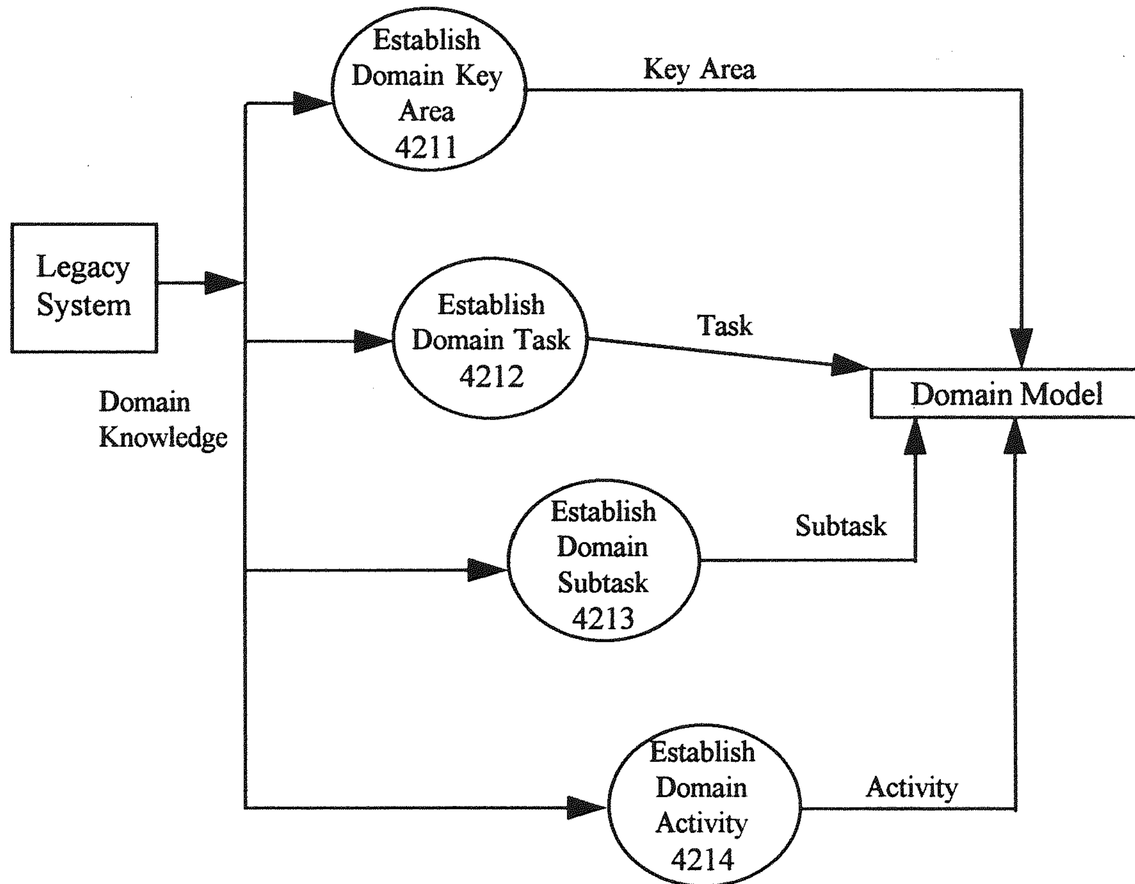
4.1.3 Select Functional Analyst. Functional analysts are selected from the organization's staff of existing system users and domain specialists. Depending on the size of the system to be reverse engineered, between two and four functional analysts are selected. Recommendations from managers and co-workers are solicited to identify the most highly qualified individuals.

4.1.4 Select Technical Analyst. A technical analyst who has knowledge and experience with the legacy system and its operating environment is selected for the reverse engineering team. Recommendations are solicited from technical managers and co-workers to identify the most highly qualified technician. In many cases, the person responsible for maintaining the system is the most qualified candidate.

4.1.5 Prepare Modeling Schedule. A schedule for the domain modeling sessions is established and coordinated with modeling team members. Working sessions are scheduled for four-hour periods on alternate days. Functional users are able to provide more accurate and more detailed input when there is time between sessions to consider previous model input.



DFD 4.2 - Model functional domain



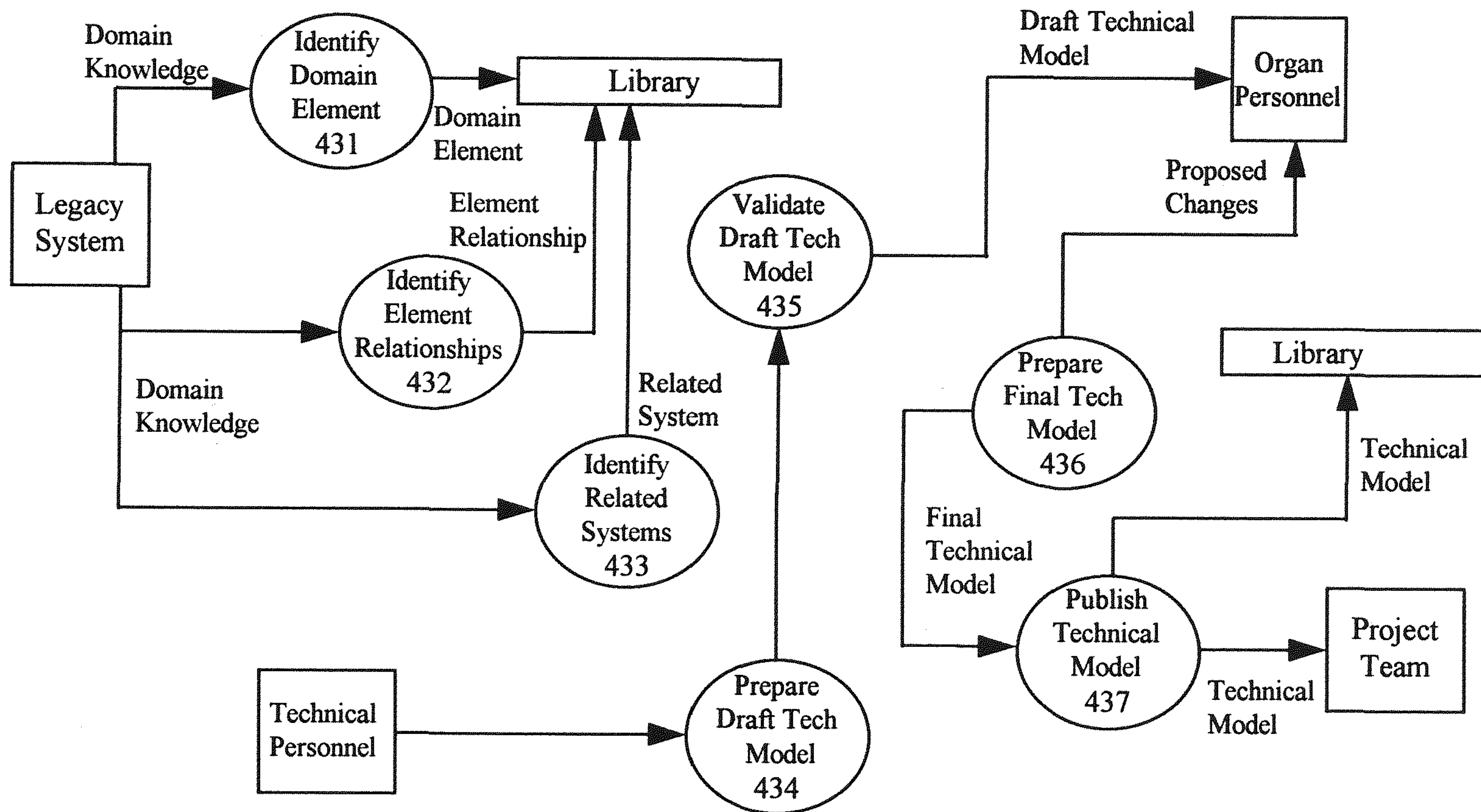
DFD 4.2.1 - Create outline model

4.2.1.1 Establish Domain Key Area. The first step in developing the domain model is identifying the major functional areas (key areas) represented in the legacy system. Five to nine key areas are identified. Dependent on the magnitude of the domain, each of the key areas could be considered as individual domain model targets (i.e., a separate model is created for each key area function). The titles for all functions in the process model are specified in the format verb + adjective + direct object. Descriptions of the functions are not written until the structure is finalized.

4.2.1.2 Establish Domain Task. Within each domain key area, five to nine tasks required to perform the key area are identified.

4.2.1.3 Establish Domain Subtask. A subtask represents an intermediate decomposition level between domain tasks and bottom-level activities and identifies the major actions required to complete a domain task. Dependent on the complexity of the domain, multiple subtasks may be identified.

4.2.1.4 Establish Domain Activity. Within each domain or subtask, five to nine activities required to perform the task or subtask are identified. An activity is the lowest level function in the domain hierarchy (i.e., a primitive function). An activity is normally defined as an independent unit of work carried out by a single individual.



DFD 4.3 - Model technical domain

4.2.2 Validate Outline Domain Model. The completed outline domain model is distributed to other functional users for review and comment. This review ensures that all users have the opportunity to provide input.

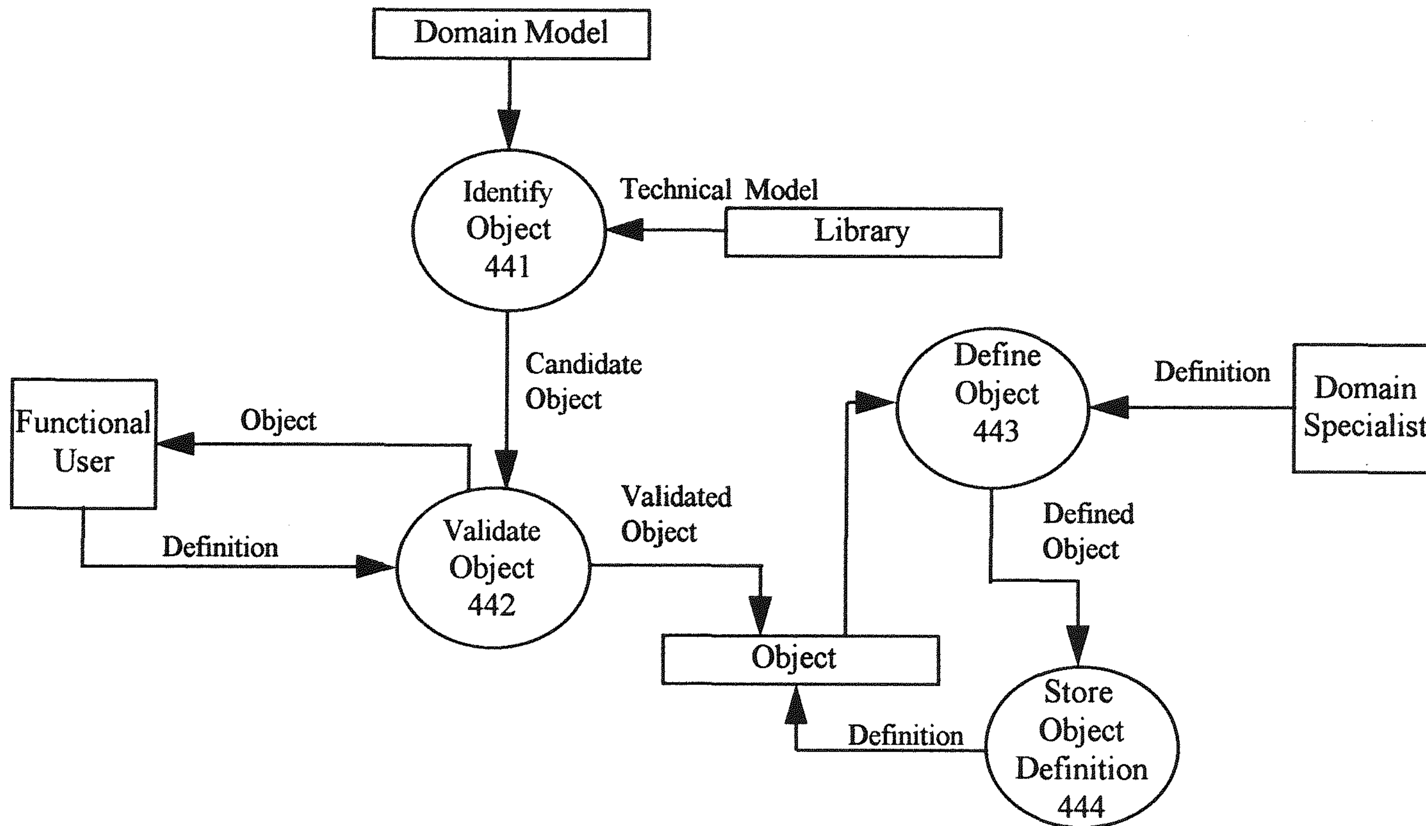
4.2.3 Revise Outline Domain Model. Recommended changes are discussed, and the outline domain model is revised by the original modeling team.

4.2.4 Create Described Domain Model. A described domain model is created by writing a narrative description for each *activity* in the outline process model. Narrative descriptions are written by functional users and domain specialists on the modeling team and are limited to a few sentences. Details of who performs the function and how it is carried out are scrupulously avoided.

4.2.5 Validate Domain Model. The described domain model is coordinated with other organizational users, domain specialists, and managers to ensure the model is comprehensive, correct, and understandable.

4.2.6 Prepare Final Domain Model. The original model developers consider each reviewer recommendation for incorporation into the domain model, and the final domain model is prepared.

4.2.7 Publish Domain Model. The completed domain model is published and made available to members of the reverse engineering team. The domain model structure is also stored in the RE repository.



DFD 4.4 - Model domain object

4.3.1 Identify Major Domain Element. Major elements of the technical domain in which the legacy system operates are identified by the functional technician and domain specialists. Elements include other automated systems, operating locations, major inputs, major output products, and primary customers.

4.3.2 Identify Element Relationship. Relationships between major elements of the technical model are identified when these relationships are significant to legacy system understanding.

4.3.3 Identify Related System. Related systems, especially those providing input to or receiving output from the target legacy system, are identified and described.

4.3.4 Prepare Draft Technical Model. A technical model summarizing the environment and systems related to the target legacy system is prepared in graphic and narrative form.

4.3.5 Validate Draft Technical Model. The draft technical model is coordinated with functional users, domain specialists, and other technicians.

4.3.6 Prepare Final Technical Model. Changes recommended during the draft technical model review are incorporated, and the final technical model is prepared.

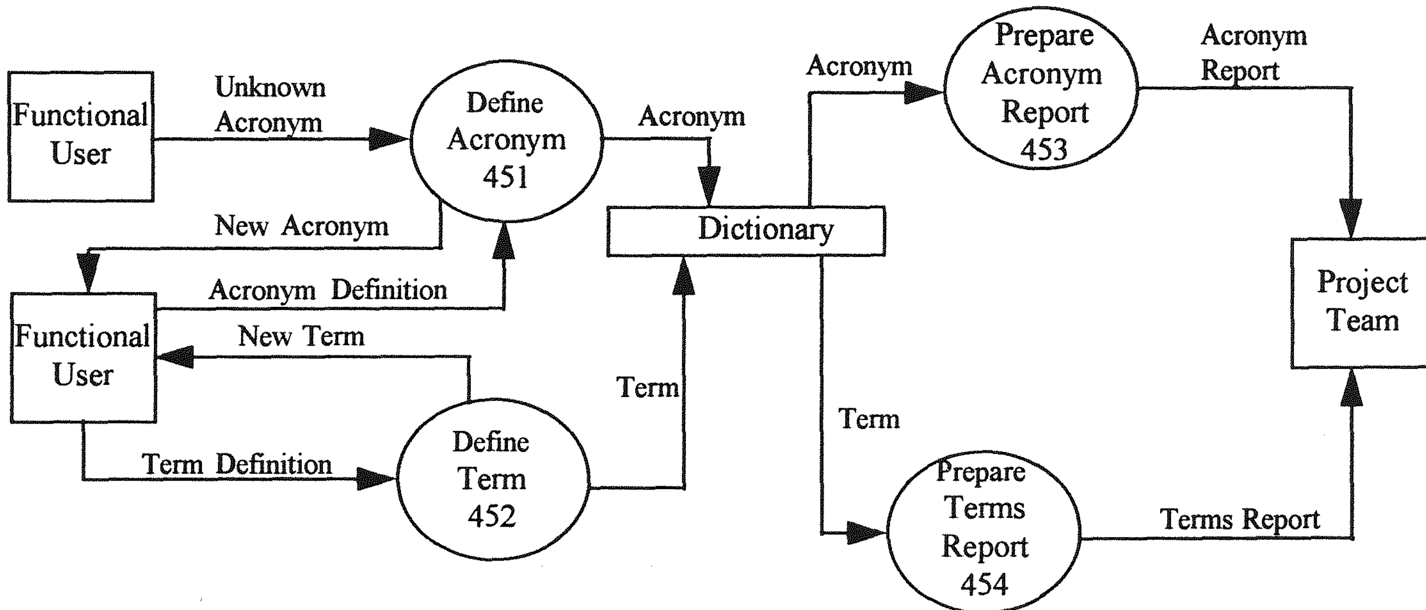
4.3.7 Publish Technical Model. The final technical model is published and provided to all members of the reverse engineering team.

4.4.1 Identify Object. A domain object captures a semantic primitive within the application domain. Candidate objects are identified from the functional domain narrative descriptions and the technical model.

4.4.2 Validate Object. Candidate domain objects are validated by functional users and domain specialists.

4.4.3 Define Object. Valid domain objects are defined with the assistance of domain specialists.

4.4.4 Store Object Definition. Object definitions are stored in a central repository for access by the reverse engineering team.



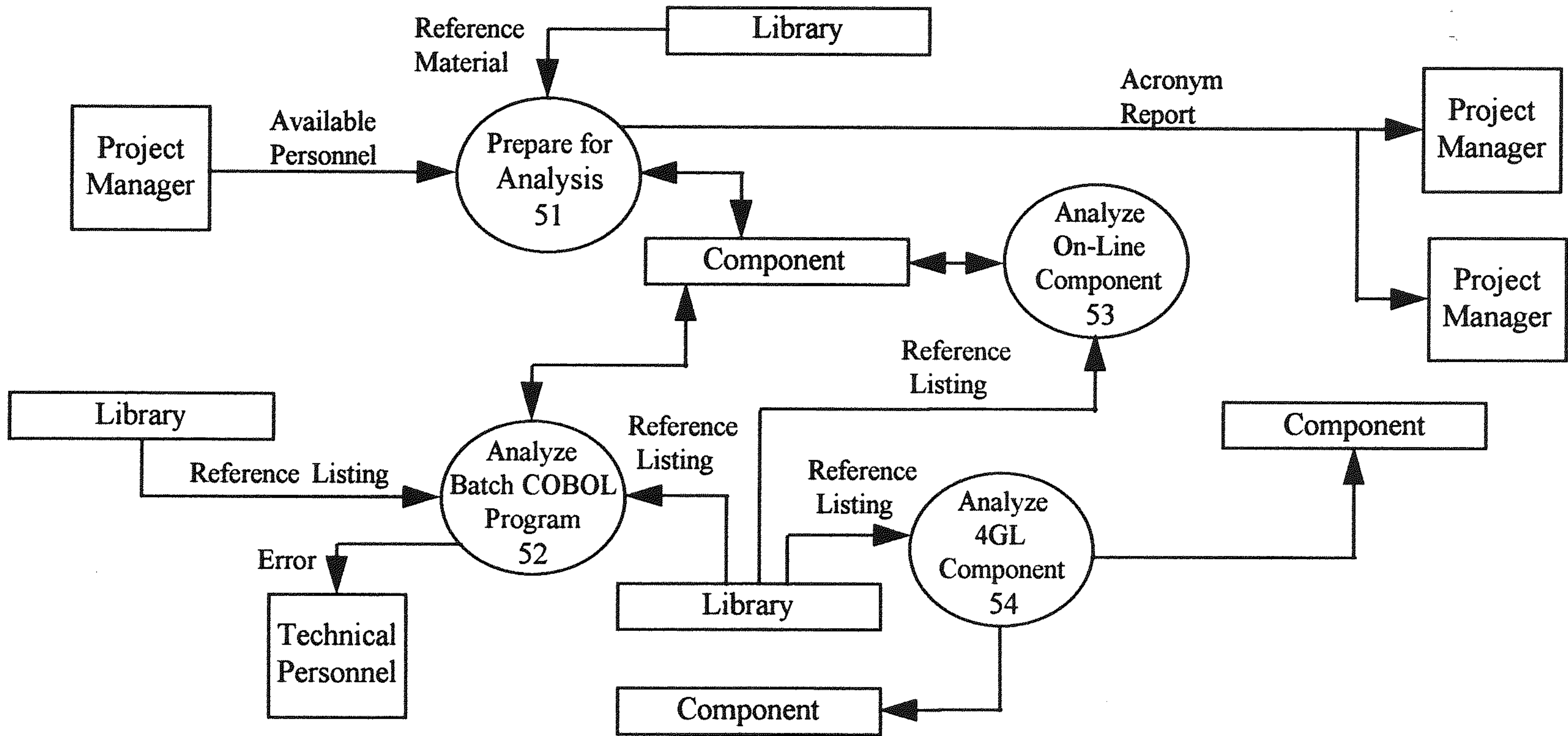
DFD 4.5 - Establish project dictionary

4.5.1 Define Acronym. Acronyms encountered during the reverse engineering effort are defined in an RE dictionary. Domain specialists validate the acronym's meaning before the acronym is placed in the dictionary.

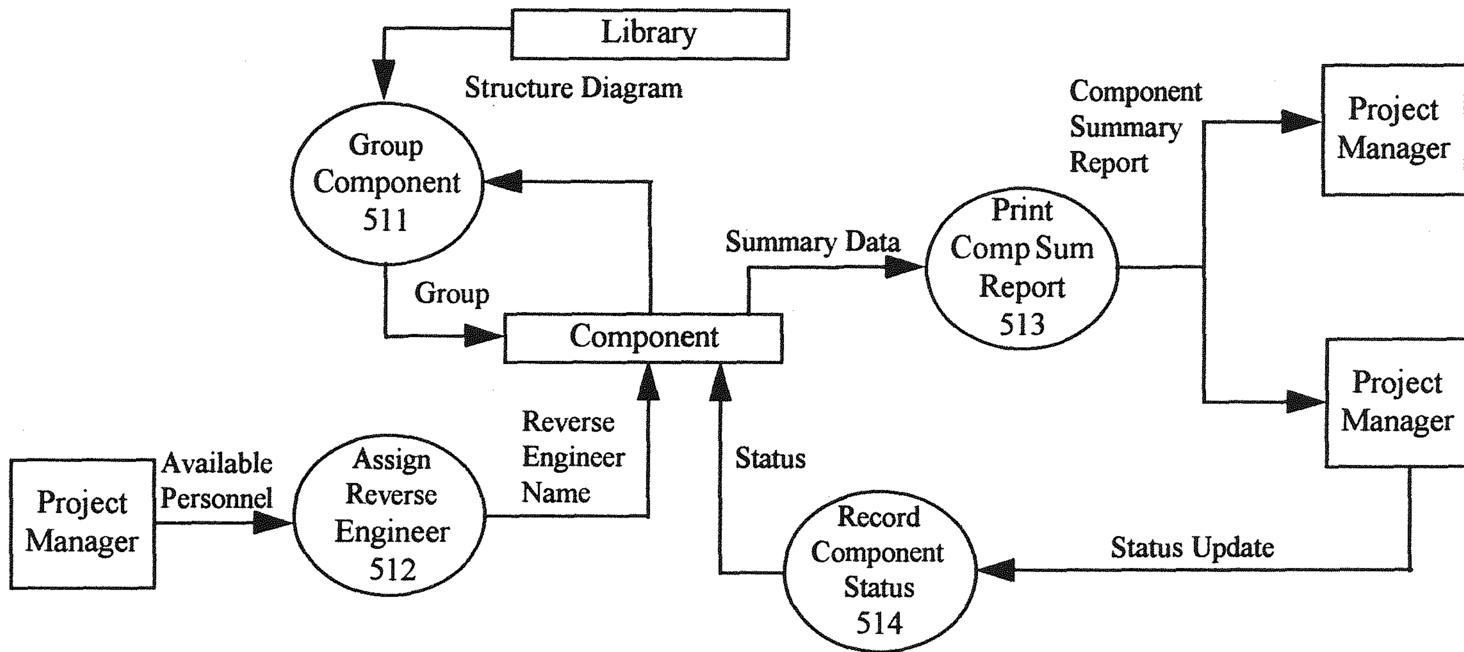
4.5.2 Define Term. Special terms encountered during the reverse engineering effort are defined in a RE dictionary. Domain specialists or functional technicians validate definitions before they are stored in the dictionary.

4.5.3 Prepare Acronym Report. A list of acronyms with their authenticated definitions is periodically prepared and distributed to reverse engineering team members.

4.5.4 Prepare Term Report. A list of special terms with their authenticated definitions is periodically prepared and distributed to reverse engineering team members.



DFD 5 - Analyze source code



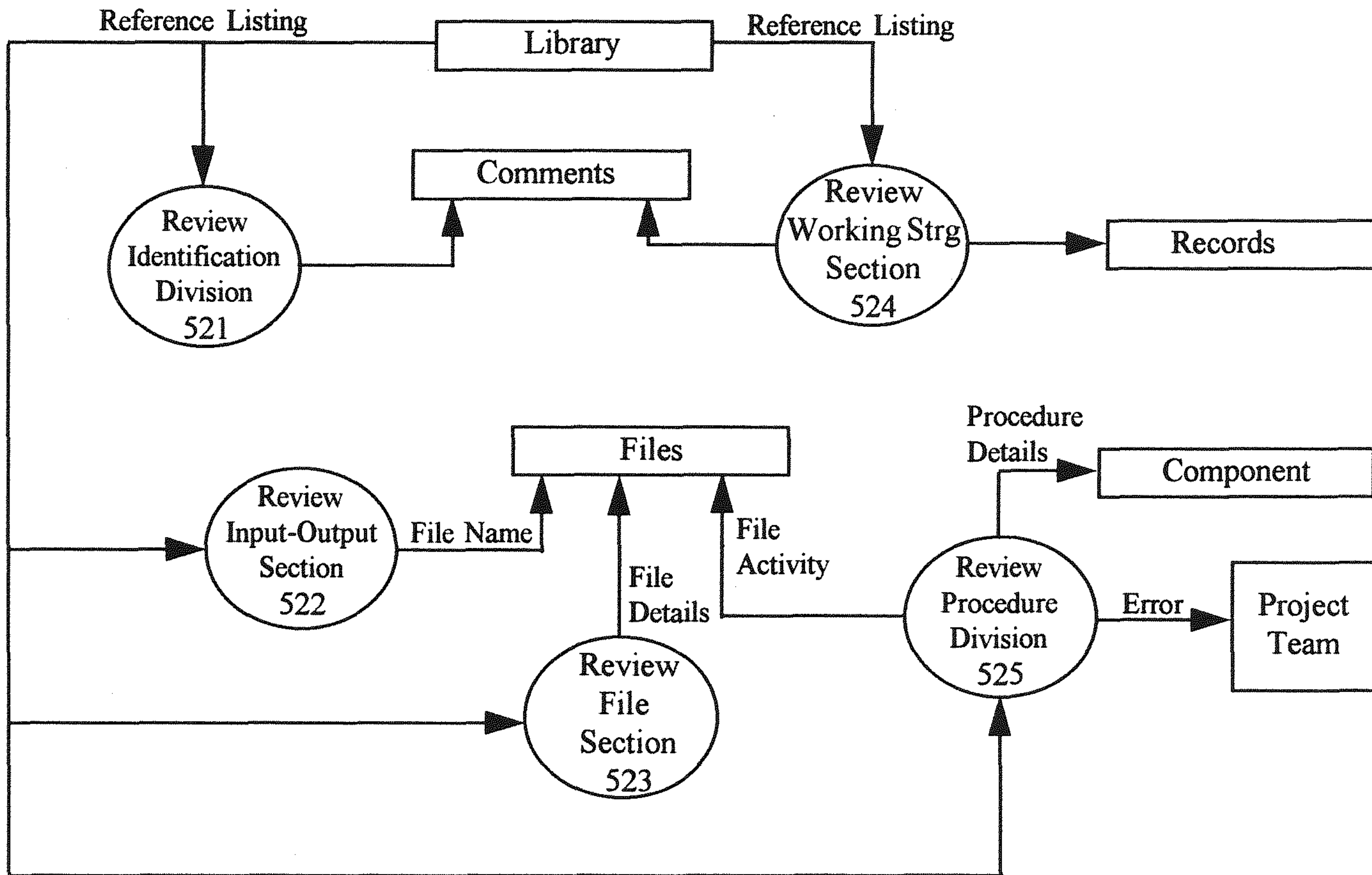
DFD 5.1 - Prepare for analysis

5.1.1 Group Component. Target system components are grouped according to subsystems or job steps to simplify reverse engineering. For efficiency, a reverse engineer should be responsible for all the programs in a group.

5.1.2 Assign Reverse Engineer. A reverse engineer responsible for creating the program model and recovering design information is assigned by name. The responsible reverse engineer's name is recorded in the RE repository.

5.1.3 Print Component Summary Report. A summary report containing component information collected during the preliminary review is printed and sent to the reverse engineering project manager and the responsible reverse engineer. The summary report is a tasking directive.

5.1.4 Record Component Status. The status of the reverse engineering effort for a component is recorded in the RE repository. Includes date assigned to reverse engineer, date reverse engineering started, expected completion date, percent completed, and actual time required to complete. The responsible reverse engineer is responsible for periodically updating the status.



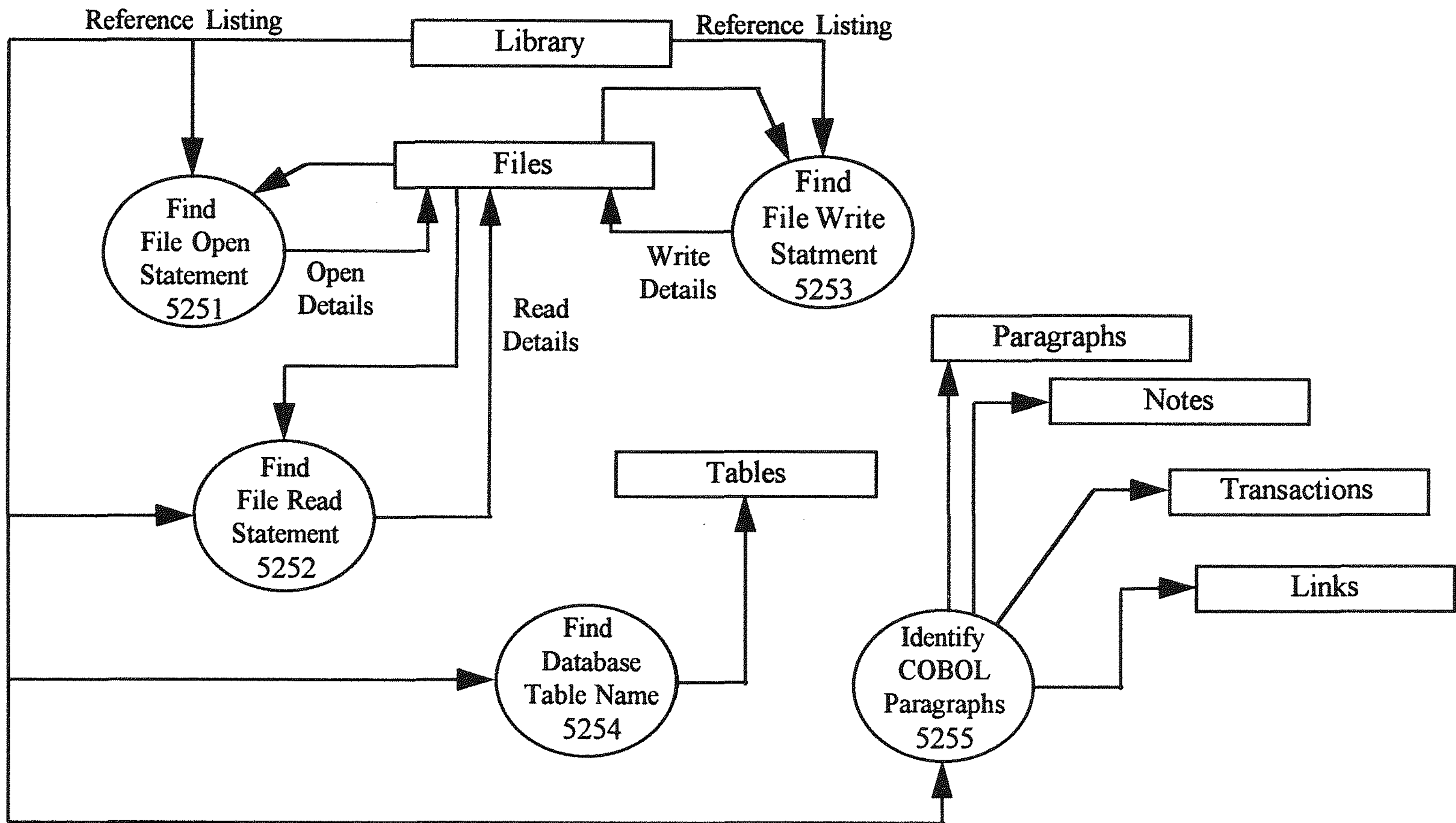
DFD 5.2 - Analyze batch COBOL component

5.2.1 Review Identification Division. The Identification Division is reviewed for informative comments. Significant comments are extracted and stored with the component description in the RE repository along with the source code line number. The source code line number is used to point to a specific location in a component if later verification or review is required.

5.2.2 Review Input-Output Section. The Environment Division Input-Output Section is reviewed to identify file SELECT statements that identify internal and external file names. File information is extracted and recorded in the RE repository.

5.2.3 Review File Section. The Data Division File Section is reviewed for FD statements for each file used by the component. FD entries identify records associated with an input or output file. Multiple record types may be specified for a file. Records for each file are recorded in the RE repository.

5.2.4 Review Working Storage Section. The Working Storage Section is reviewed to identify database tables used, record formats, and other significant data structures or in-line comments. Tables used by the component are recorded in the RE repository.



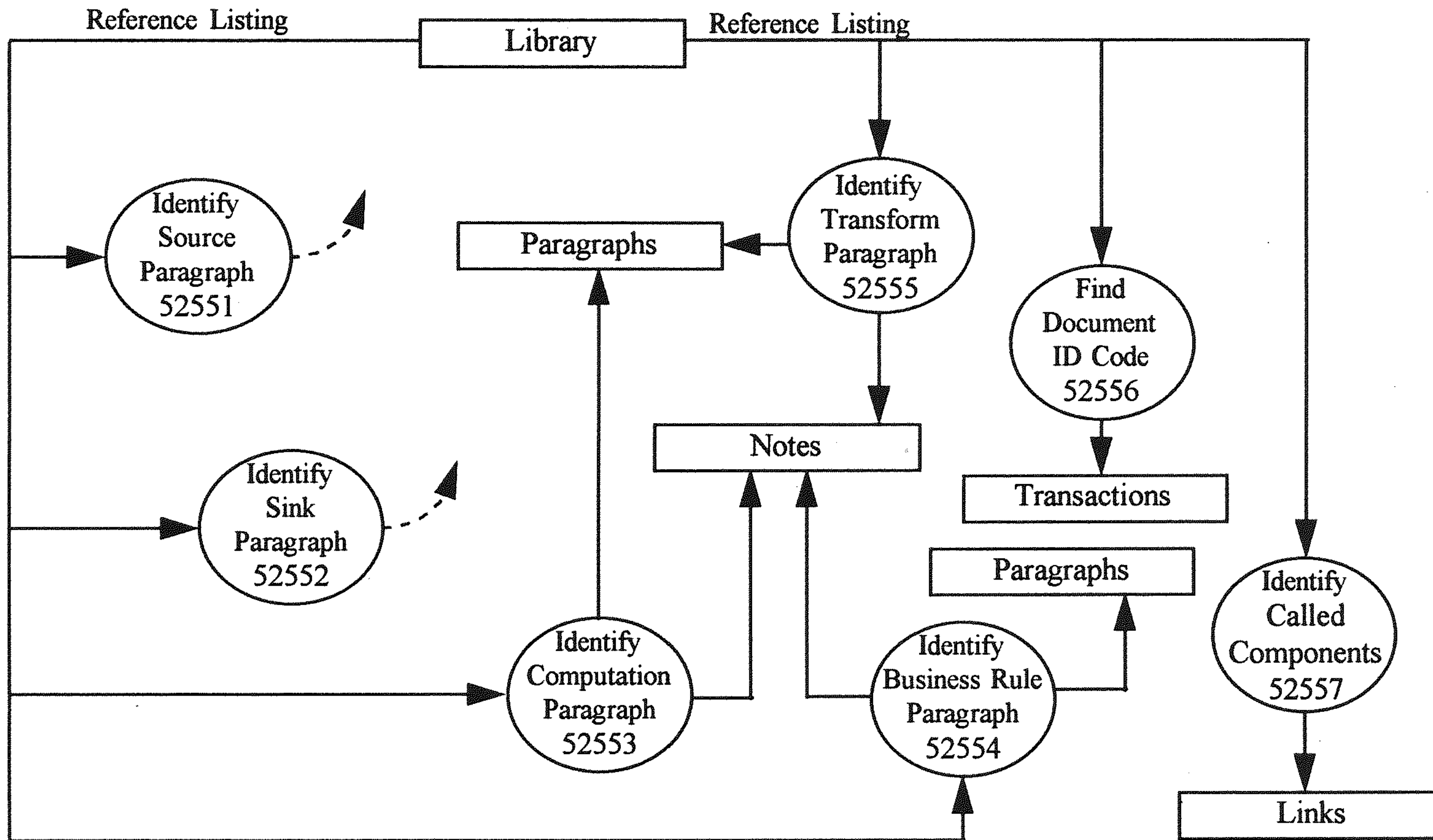
DFD 5.2.5 - Review COBOL procedure division.

5.2.5.1 Find File OPEN Statement. The Procedure Division is scanned to find an OPEN statement for each file identified in a SELECT statement. The OPEN statement indicates how a file is used (e.g., input, output, or I-O).

5.2.5.2 Find File READ Statement. The Procedure Division is scanned to find READ statements for each file accessed as input by the component. Record formats associated with the file are identified and stored in the RE repository.

5.2.5.3 Find File WRITE Statement. The Procedure Division is scanned to find WRITE statements for each file accessed as output by the component. Record types written to the file are identified and stored in the RE repository.

5.2.5.4 Find Database Table Name. The Procedure Division is scanned to locate each database table accessed by the component. Activity with respect to the table (create, read, update, or delete) is recorded in the RE repository.



DFD 5.2.5.5 - Identify COBOL paragraph

5.2.5.5.1 Identify Source Paragraph. A source paragraph is an environment-dependent module and is ignored for abstraction purposes.

5.2.5.5.2 Identify Sink Paragraph. A sink paragraph is an environment-dependent module and is ignored for abstraction purposes.

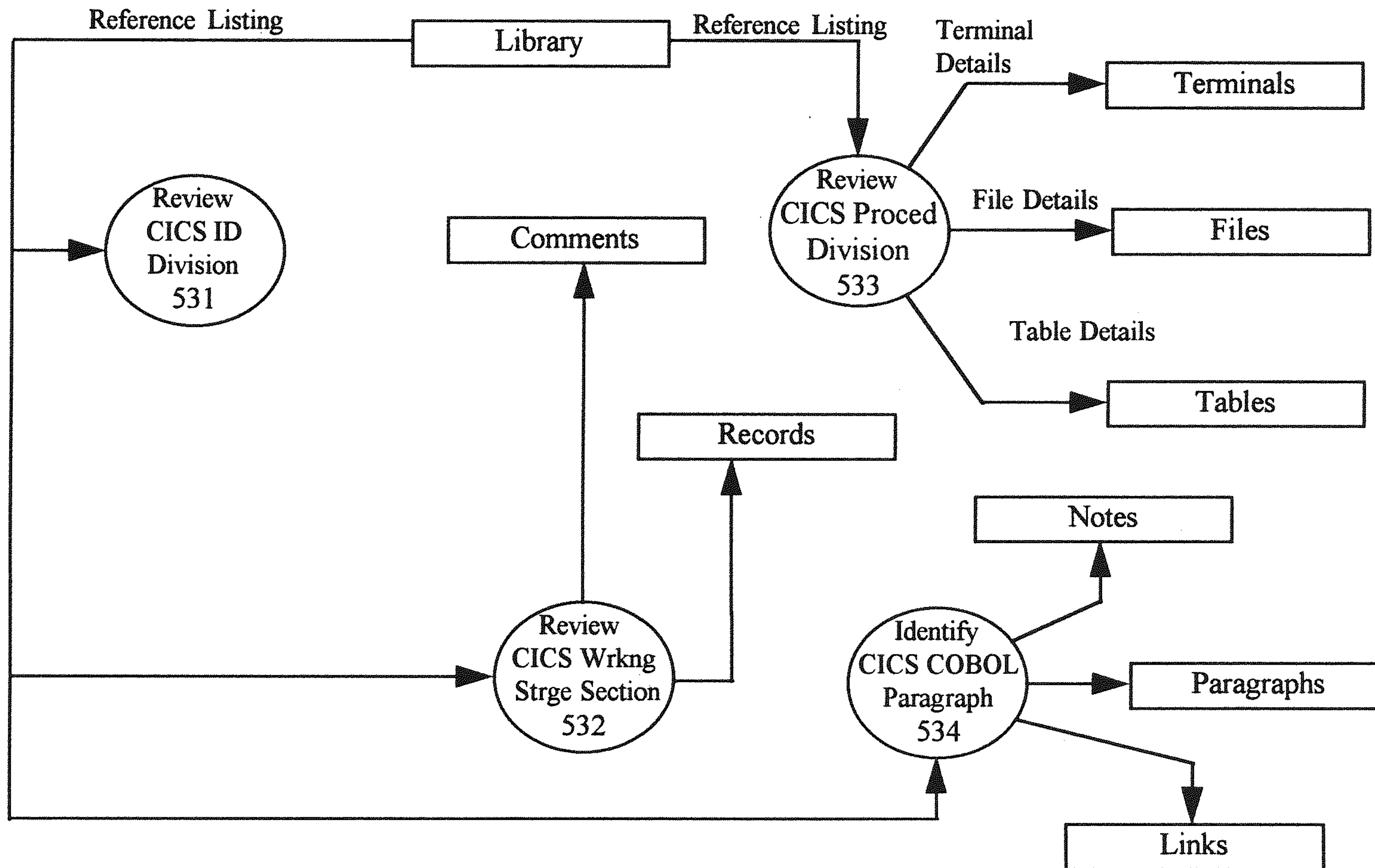
5.2.5.5.3 Identify Computation Paragraph. A paragraph containing a computation formula is a domain-dependent paragraph and is extracted from the component. The paragraph name, locating line number, in-line comment (if included), and a note summary of the paragraph are extracted and stored in the RE repository.

5.2.5.5.4 Identify Business Rule Paragraph. A paragraph containing an identifiable business rule is a domain-dependent paragraph. The paragraph name, locating line number, in-line comment (if included), and a note summary of the paragraph are extracted and stored in the RE repository.

5.2.5.5.5 Identify Transform Paragraph. A transform paragraph (i.e., one that is not a source, sink, computation, or business rule) is a domain-dependent function and is extracted from the component. The paragraph name, locating line number, and in-line comment (if included) are recorded in the RE repository.

5.2.5.5.6 Find Document Identifier Code (DIC). The Procedure Division is scanned to find DIC used in the component. DIC, descriptions, and activity (create, read, update, or delete) are stored in the RE repository.

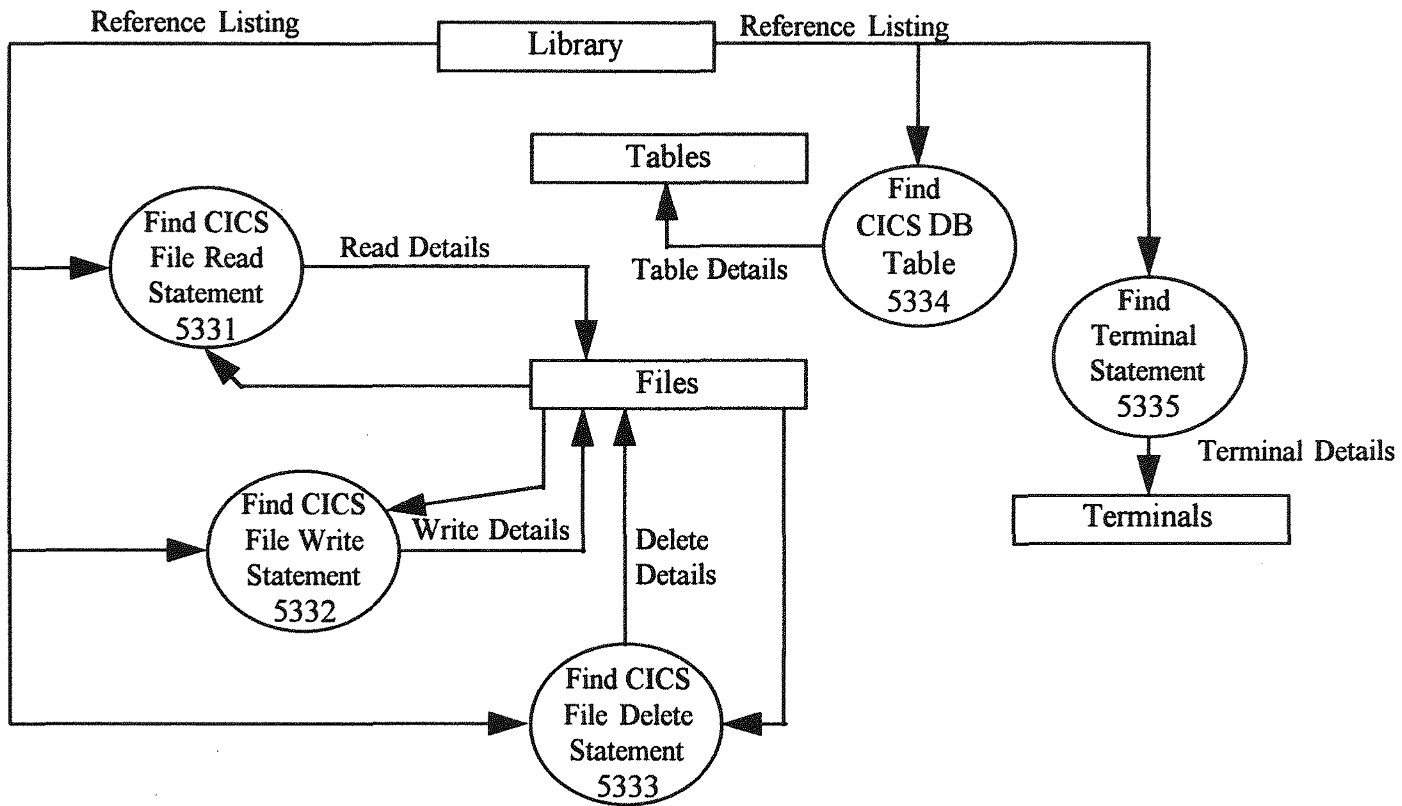
5.2.5.5.7 Identify Called Component. The Procedure Division is scanned to find CALL statements. The name in single quotation marks following CALL is the subprogram name. Parameters, if used, are recorded in the RE repository.



DFD 5.3 - Analyze on-line component

5.3.1 Review CICS Identification Division. The Identification Division of a CICS component is reviewed for informative comments explaining the program. Significant comments are extracted and stored in the RE repository along with the locating line number.

5.3.2 Review CICS Working Storage Section. The Working Storage Section of a CICS component is reviewed to find database tables used, transaction formats, and other significant data structures or in-line comments. Tables used by the component, comments, and locating line numbers are stored in the RE repository.



DFD 5.3.3 - Review CICS procedure division

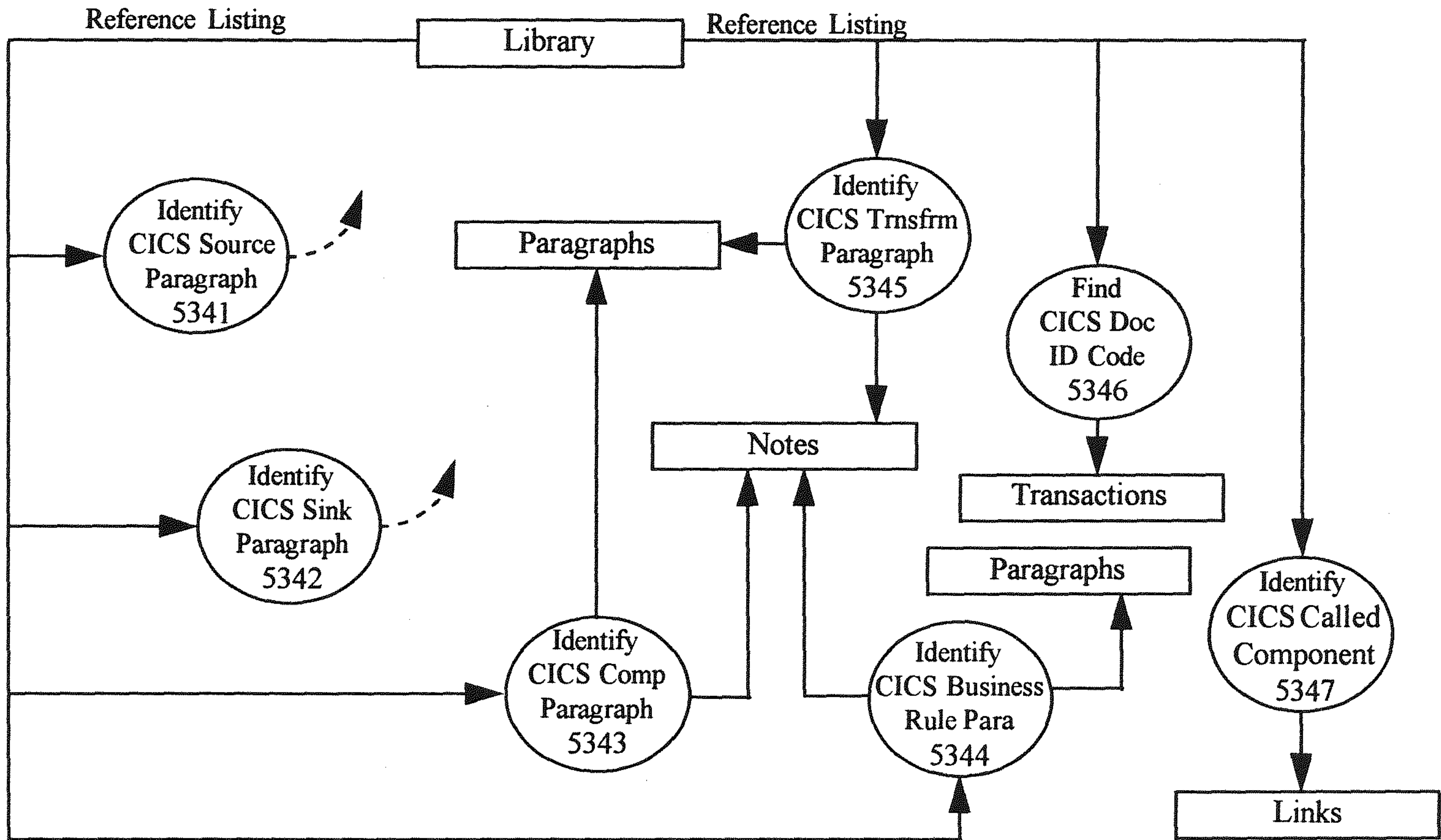
5.3.3.1 Find CICS File Read Statement. The Procedure Division of a CICS component is scanned to find virtual storage access method (VSAM) file read statements. Entry sequenced data sets (ESDS), keyed sequential data sets (KSDS), or relative record data sets (RRDS) may be used. File record layouts and activity (read or update) are stored in the RE repository.

5.3.3.2 Find CICS File Write Statement. The Procedure Division of a CICS component is scanned to find file write statements. ESDS, KSDS, or RRDS files may be used. Record layouts and file activity (i.e., create, update) are stored in the RE repository.

5.3.3.3 Find CICS File Delete Statement. The Procedure Division of a CICS component is scanned to find file delete statements. KSDS and RRDS files may be used. Record layout and file activity (delete) are stored in the RE repository.

5.3.3.4 Find CICS Database Table. The Procedure Division of a CICS component is scanned to find each database table accessed by the component and to determine the activity with respect to the table (create, read, update, or delete). The database table name and activity are stored in the RE repository.

5.3.3.5 Find Terminal Statement. The Procedure Division of a CICS component is scanned to locate input and output associated with on-line terminals. Significant data elements or data structures are identified and stored in the RE repository.



DFD 5.3.4 - Identify CICS paragraph

5.3.4.1 Identify CICS Source Paragraph. A CICS component source paragraph is an environment-dependent module and is ignored for abstraction purposes.

5.3.4.2 Identify CICS Sink Paragraph. A CICS component sink paragraph is an environment-dependent module and is ignored for abstraction purposes.

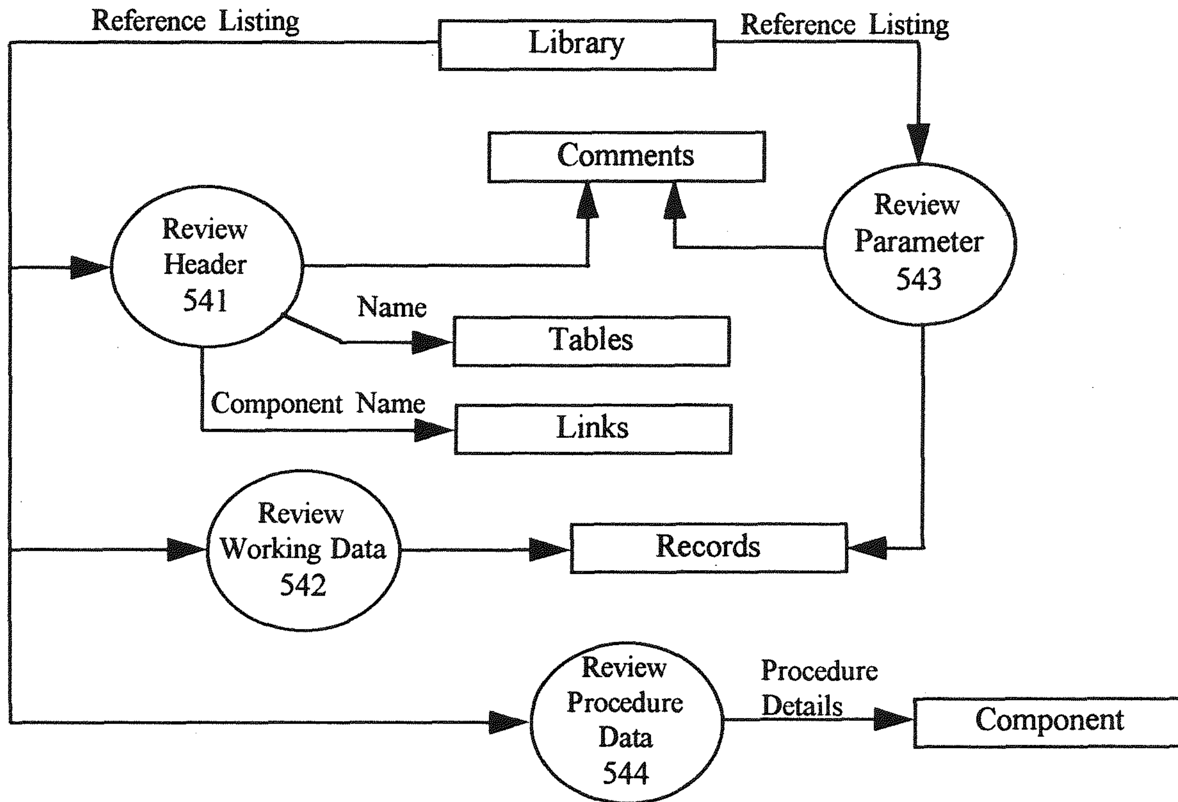
5.3.4.3 Identify CICS Computation Paragraph. A CICS component paragraph containing a computation formula is a domain-dependent paragraph and is extracted from the component. The paragraph name, locating line number, in-line comment (if included), and a note summary of the paragraph are extracted and stored in the RE repository.

5.3.4.4 Identify CICS Business Rule Paragraph. A CICS component paragraph containing an identifiable business rule is a domain-dependent paragraph. The paragraph name, locating line number, in-line comment (if included), and a note summary of the paragraph are extracted and stored in the RE repository.

5.3.4.5 Identify CICS Transform Paragraph. A CICS component transform paragraph (i.e., one that is not a source, sink, computation, or business rule) is a domain-dependent function and is extracted from the component. The paragraph name, locating line number, and in-line comment (if included) are recorded in the RE repository.

5.3.4.6 Find CICS Document Identifier Code (DIC). The Procedure Division of a CICS program is scanned to find DIC used in the component. DIC, descriptions, and activity (create, read, update, or delete) are stored in the RE repository.

5.3.4.7 Identify CICS Called Component. The Procedure Division of a CICS component is scanned to find EXEC CICS LINK or EXEC CICS XCTL commands (bi-directional and uni-directional calls, respectively). The file identification in the PROGRAM option is the link-to program name. The source code is analyzed to determine the generic data passed to the link-to program through the communication work area. Linking details are stored in the RE repository.

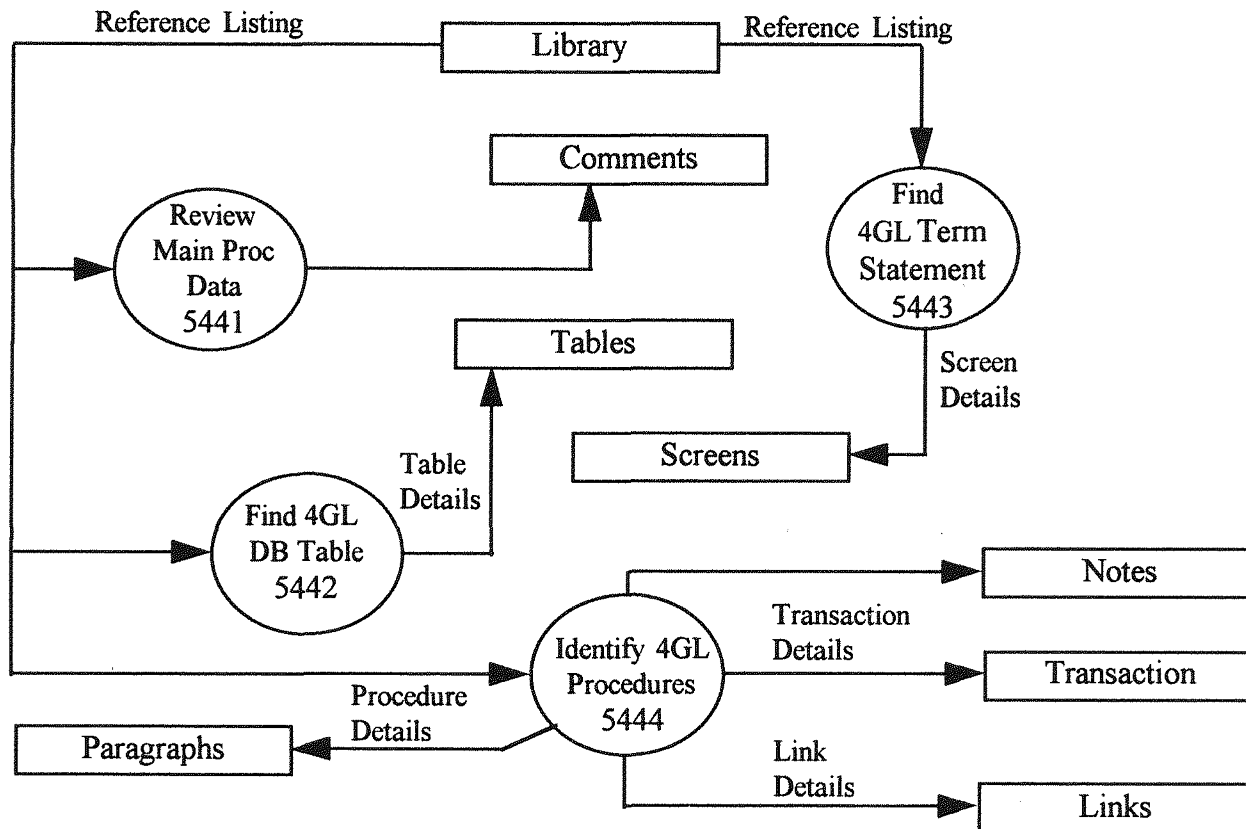


DFD 5.4 - Analyze 4GL component

5.4.1 Review Header. The header of a 4GL component is scanned for informative comments, data tables used, and programs used (called). Significant comments, table names, and program names used are stored in the RE repository. Locating line numbers for comments (assigned when the source code was printed) are also recorded.

5.4.2 Review Working-Data. The Working-Data Section of a 4GL component is reviewed to identify significant data structures, which are then stored in the RE repository.

5.4.3 Review Parameter Data. The Parameter-Data section of a 4GL component is reviewed for significant data structure or in-line comments, which are then stored in the RE repository.

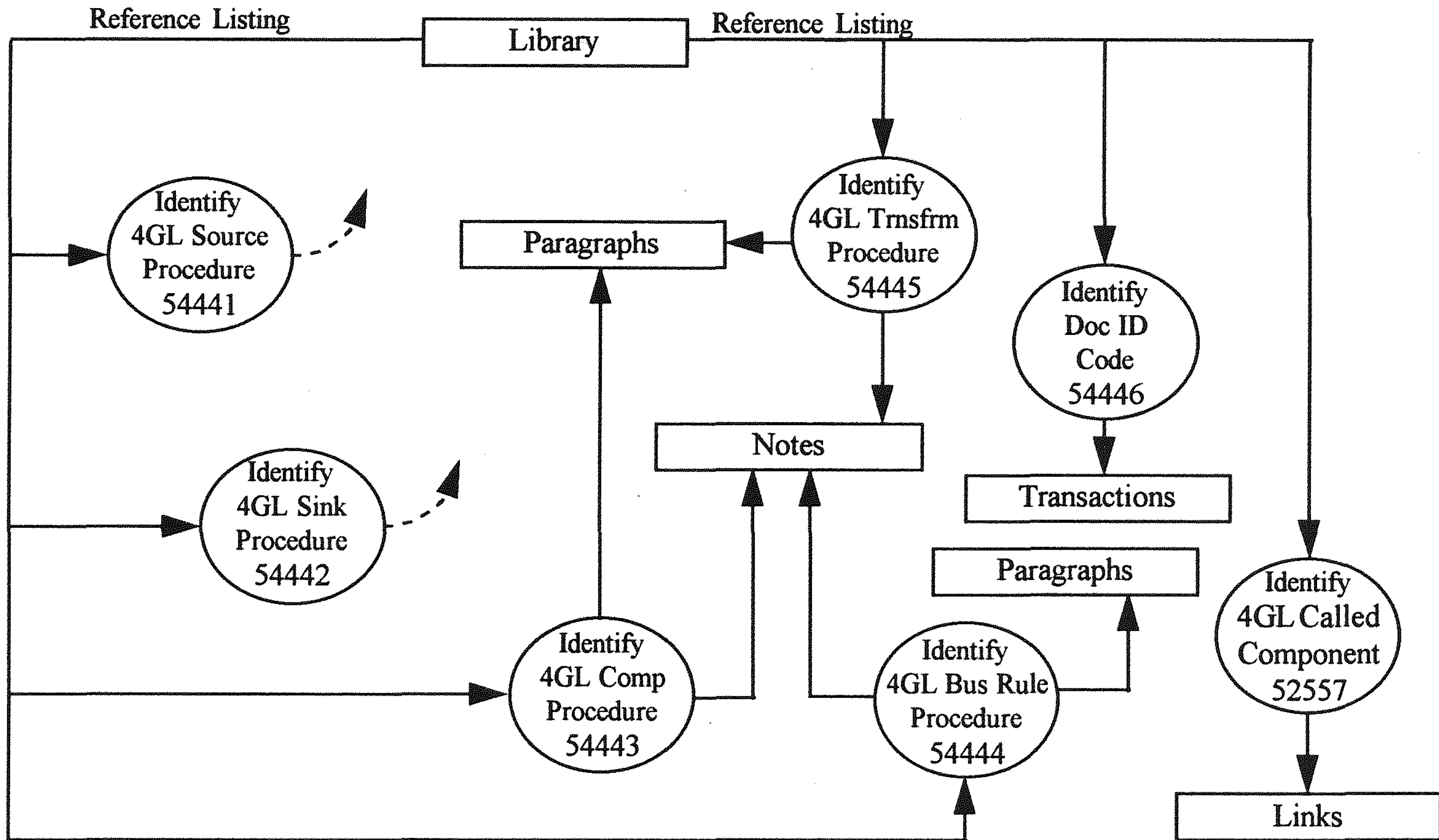


DFD 5.4 .4 - Review procedure data

5.4.4.1 Review Main Procedure Data. The Main Procedure section of a 4GL component is scanned for in-line comments. Significant comments are stored in the RE repository with locating line numbers.

5.4.4.2 Find 4GL Database Table. The main procedure and all sub-procedures in a 4GL component are scanned to find each database table accessed and to determine the table activity (create, read, update, or delete). Database tables and activity are recorded in the RE repository.

5.4.4.3 Find 4GL Terminal Statement. The main procedure and all sub-procedures in a 4GL component are reviewed to find input and output associated with an on-line terminal. Significant data elements or data structures are identified and stored in the RE repository.



DFD 5.4.4.4 - Identify 4GL procedure

5.4.4.4.1 Identify 4GL Source Procedure. A 4GL component source (input) procedure is an environment-dependent module and is ignored for abstraction purposes.

5.4.4.4.2 Identify 4GL Sink Procedure. A 4GL component sink (output) procedure is an environment-dependent module and is ignored for abstraction purposes.

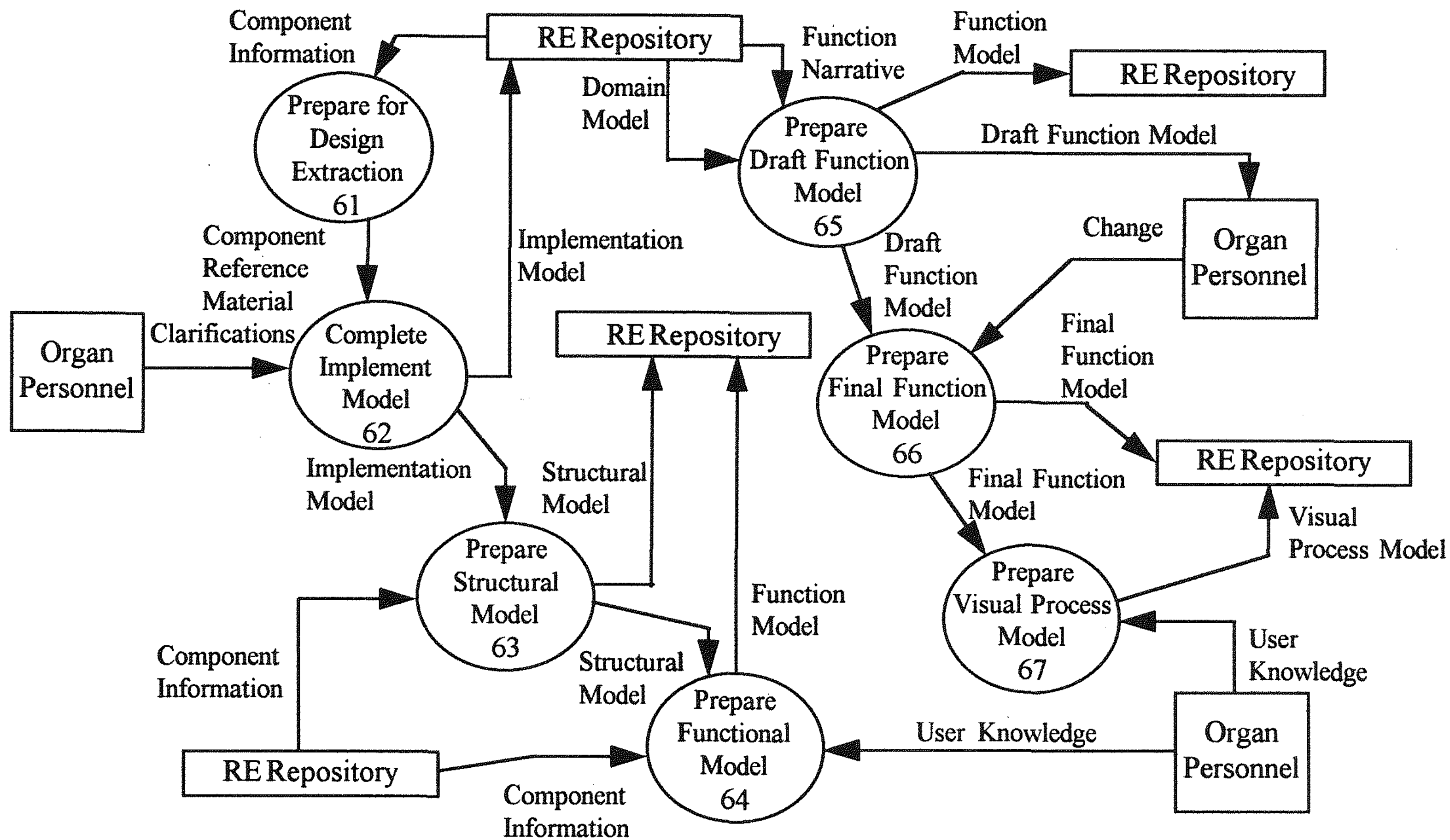
5.4.4.4.3 Identify 4GL Computation Procedure. A 4GL component procedure containing a computation formula is a domain-dependent procedure and is extracted from the component. The procedure name, locating line number, in-line comment (if included), and a note summary of the procedure are extracted and stored in the RE repository.

5.4.4.4.4 Identify 4GL Business Rule Procedure. A 4GL component procedure containing an identifiable business rule is a domain-dependent module. The procedure name, locating line number, in-line comment (if included), and a note summary of the procedure are extracted and stored in the RE repository.

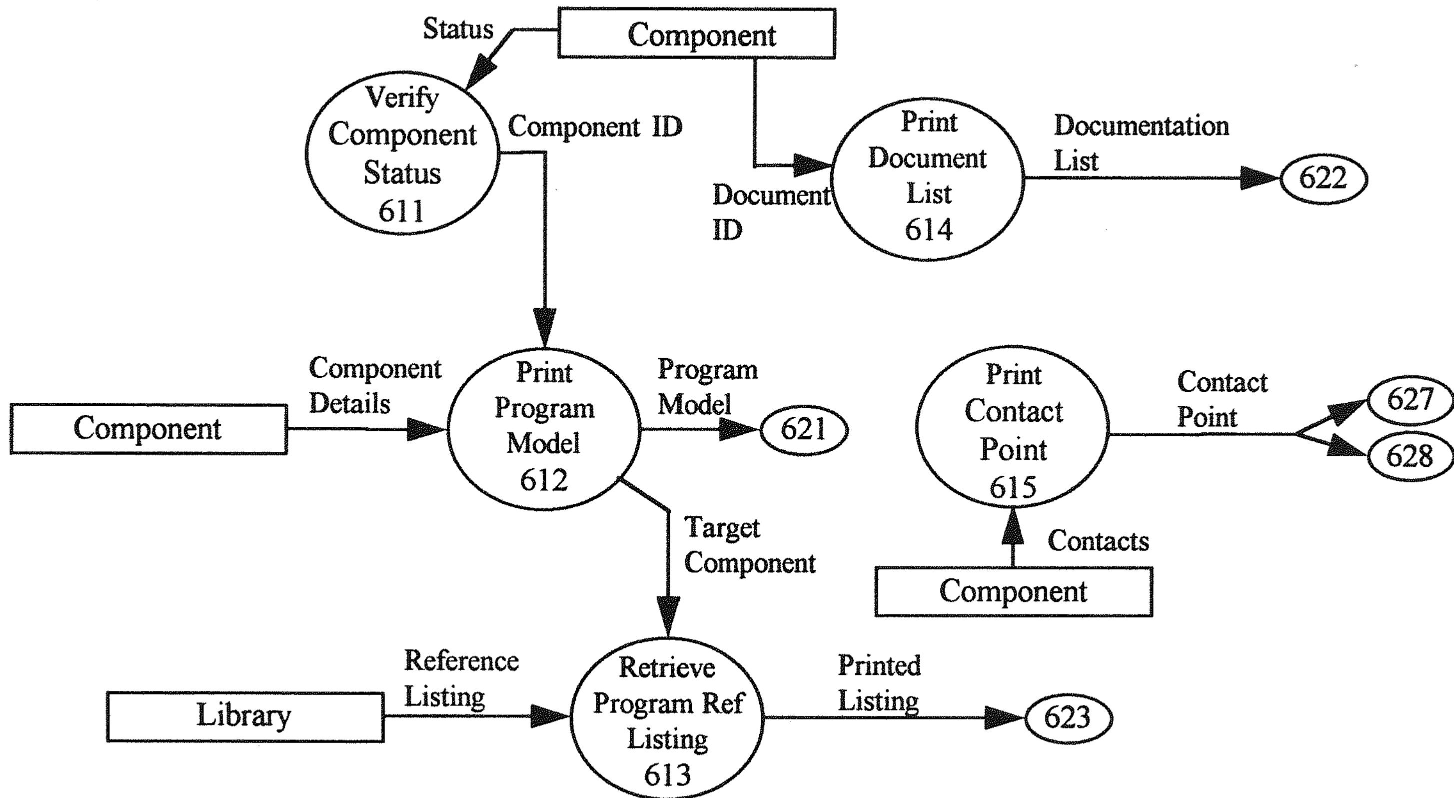
5.4.4.4.5 Identify 4GL Transform Procedure. A 4GL component transform procedure (i.e., one that is not a source, sink, computation, or business rule) is a domain-dependent module and is extracted from the component. The procedure name, locating line number, and in-line comment (if included) are recorded in the RE repository.

5.4.4.4.6 Find 4GL Document Identifier Code (DIC). The main procedure and all sub-procedures in a 4GL component are scanned to find DIC. DIC, descriptions, and activity (create, read, update, or delete) are stored in the RE repository.

5.4.4.4.7 Identify 4GL Called Component. The main and sub-procedure sections of a 4GL component are scanned to identify CALL statements. The name following the CALL statement is the subprogram name. If the USING statement follows the subprogram identification, the call passes parameters (a strong call). Data elements following the USING statement are interpreted to determine the generic data being passed to the subprogram. CALL statement details are stored in the RE repository.



DFD 6 - Extract design information



DFD 6 .1 - Prepare for design extraction

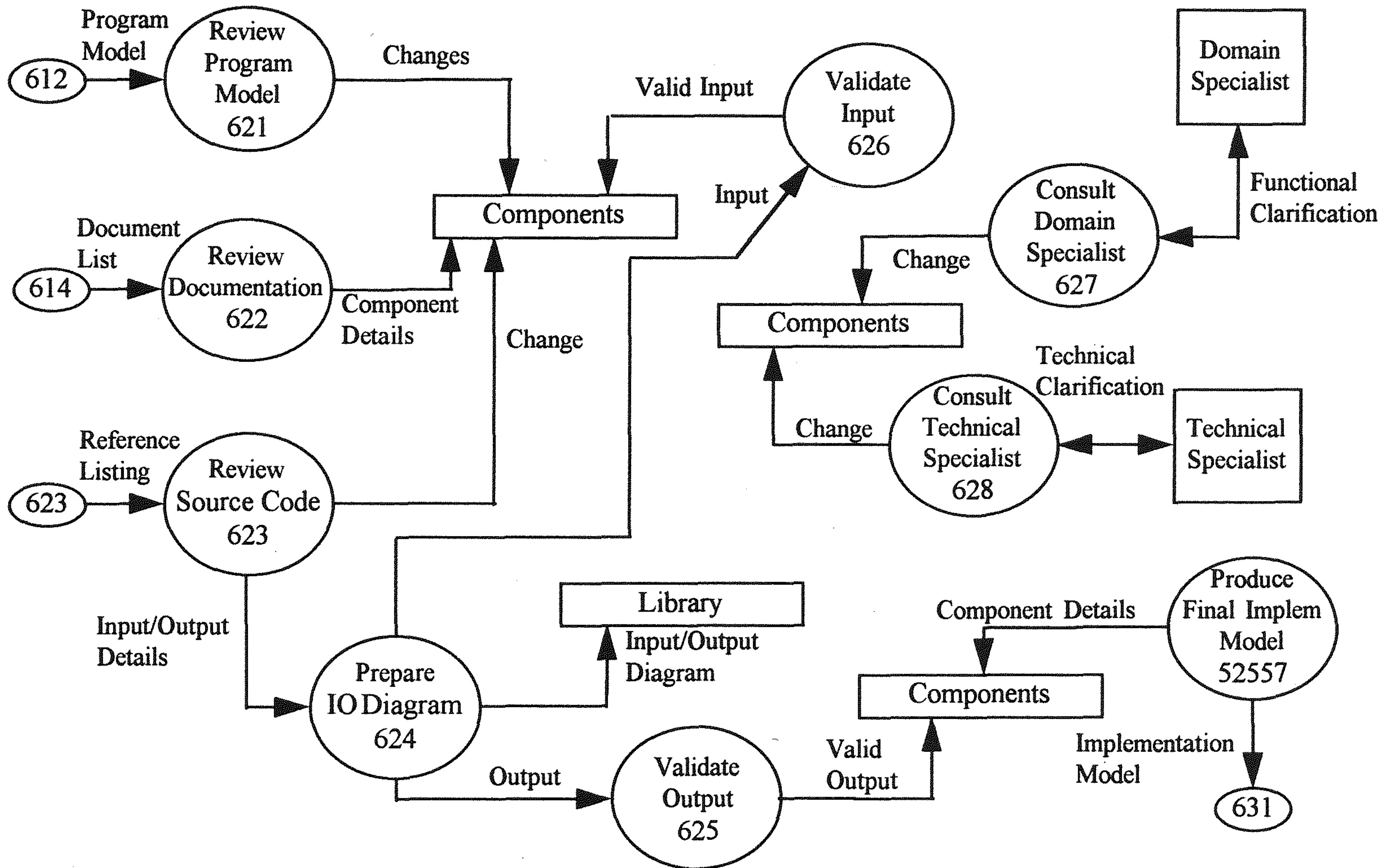
6.1.1 Verify Component Status. The status of a component is verified to ensure the preliminary review has been completed and the data necessary for design extraction is available.

6.1.2 Print Program Model. The program model developed during earlier analysis is retrieved from the RE repository, formatted, and printed.

6.1.3 Retrieve Program Reference Listing. The source code listing for a component is retrieved from the RE repository.

6.1.4 Print Documentation List. The index to documentation available for a component is retrieved from the RE repository and printed.

6.1.5 Print Contact Point. Domain specialist and technical specialist points of contract for a component are retrieved from the RE repository and printed.



DFD 6.2 - Complete implementation model

6.2.1 Review Program Model. The previously prepared program model (skeletal implementation model) is reviewed for completeness and accuracy. Changes are made if appropriate and the RE repository is updated.

6.2.2 Review Documentation. Using the documentation index as a guide, available documentation for the component is reviewed. Purpose, objectives, assumptions, and constraints for the component are identified and stored in the RE repository.

6.2.3 Review Source Code. The original component source code is reviewed for familiarization and to validate the thoroughness of the program model. If required, additions and modifications are made to the program model.

6.2.4 Prepare Input-Output Diagram. A context level data flow diagram is prepared to show all the input and output for the component. When appropriate, sources and sinks are identified. Input includes files, records, DIC, and database tables.

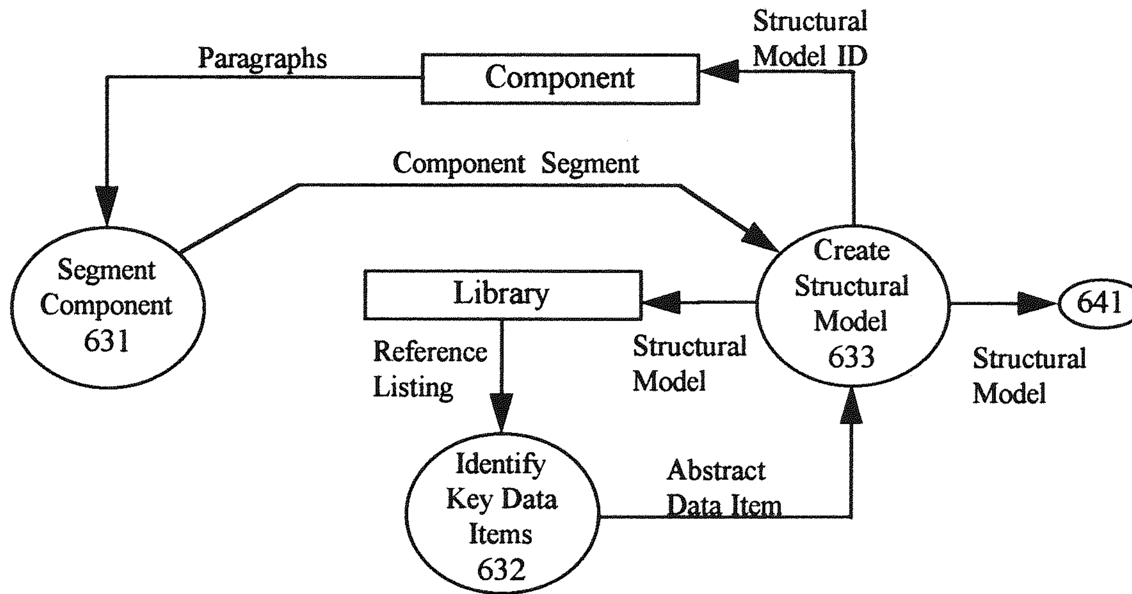
6.2.5 Validate Output. Output data streams are validated against the source code and data from the RE repository. Discrepancies are resolved by updating the model or by noting errors in source documents.

6.2.6 Validate Input. Input data streams are validated against the source code and data from the RE repository. Discrepancies are resolved by updating the model or by noting errors in source documents.

6.2.7 Consult Domain Specialist. The responsible domain specialist is consulted to resolve discrepancies found in the program model or documentation.

6.2.8 Consult Technical Specialist. The responsible technical specialist is consulted to resolve technical discrepancies found in the program model or documentation.

6.2.9 Produce Final Implementation Model. The final implementation model is produced by assembling the skeletal process structure, record layouts, screen diagrams, and other clarifying and supporting documents as appropriate.

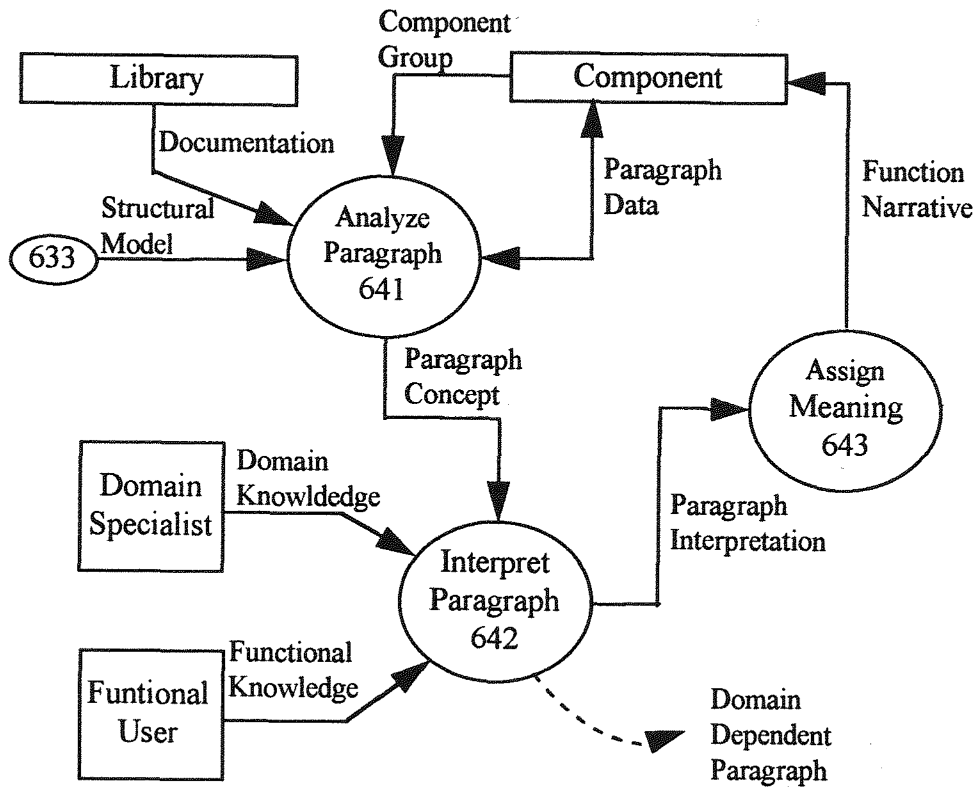


DFD 6.3 - Prepared structural model

6.3.1 Segment Component. A component is segmented into multiple areas by grouping the structural paragraphs into a logical group. Subprogram links are included in a group or shown as a separate group. This task is simplified if meaningful paragraph names and comments were used in the source code.

6.3.2 Identify Key Data Item. Key data items in each structural paragraph are identified and related to abstract, informal concepts. The focus is on data structures representing domain objects rather than on data elements describing objects.

6.3.3 Create Structural Model. A structural model is created to show the major component paragraph groups and data items.

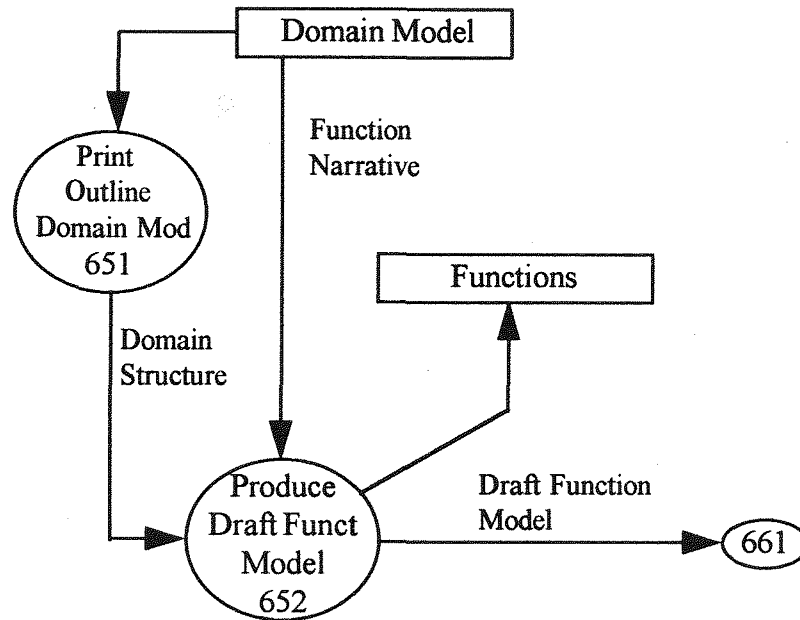


DFD 6.4 - Prepare functional model

6.4.1 Analyze Paragraph. The structural paragraphs within a component group are analyzed for understanding. If reference to original code is necessary, paragraph location numbers are used to locate full text in the source listing. Available documentation is reviewed and functional, technical, and domain specialists are consulted as required to understand each paragraph.

6.4.2 Interpret Paragraph. Individual paragraphs are interpreted to transform the source code information into functional equivalents. Domain-independent paragraphs (e.g., input and output) should have been removed from the model; if not, they are discarded. Error checking and validation routines should also be omitted; it is assumed all data is valid.

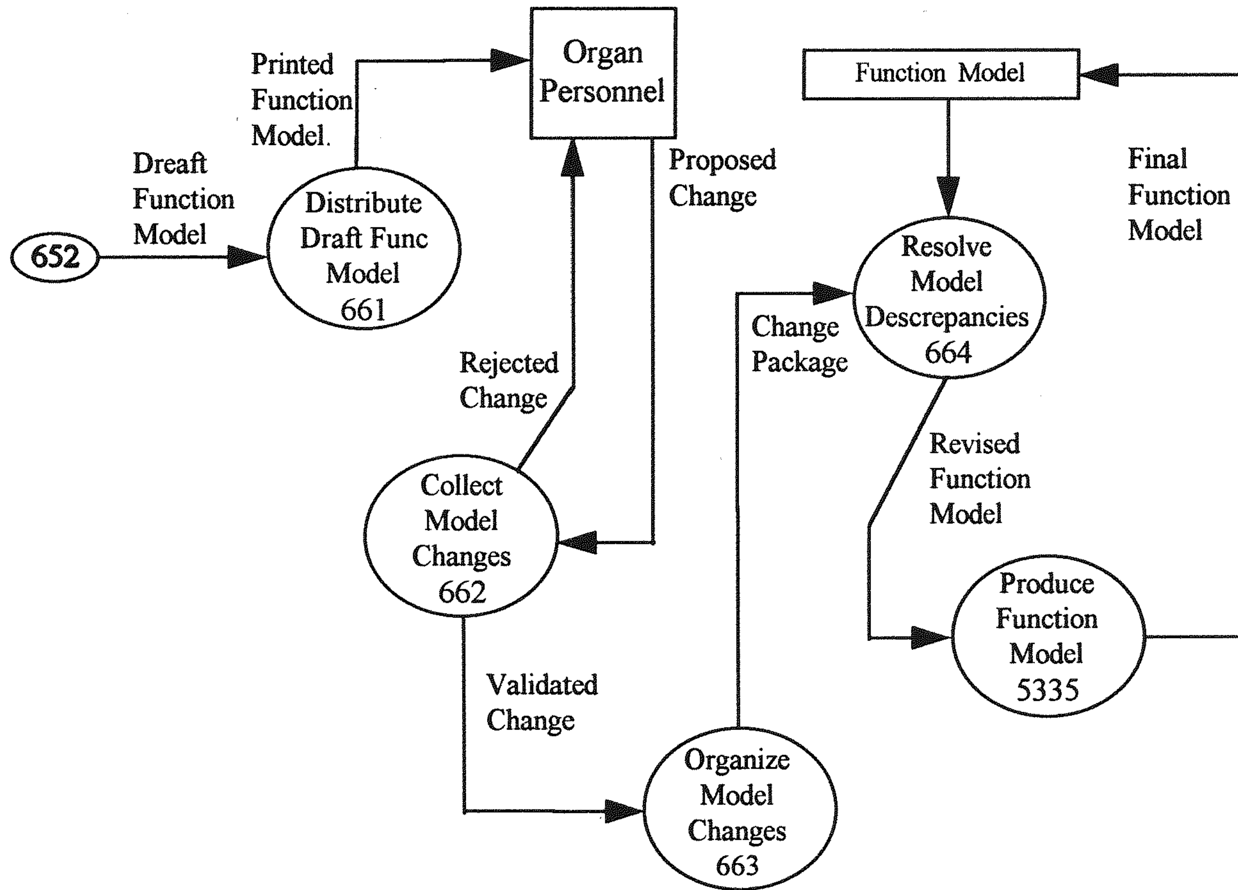
6.4.3 Assign Meaning. The computational intent of the text in each paragraph is expressed in human-oriented terms instead of technically-oriented terms. Close coordination with functional and technical analysts may be required. When completed, the description of what a paragraph does should be free of conditional statements, validation and error checking statements, and other computer or computer language concepts. Narrative should be concise and clearly written using short, simple sentences.



DFD 6.5 - Prepare draft function model

6.5.1 Print Outline Domain Model. The outline domain model previously prepared is printed for use in creating the final domain model. The outline model is printed to place each activity on a separate page to allow space to enter the reverse engineered functions.

6.5.2 Produce Draft Function Model. The draft function model is prepared by assigning each function from a program to an activity in the domain model.



DFD 6.6 - Prepare final function model

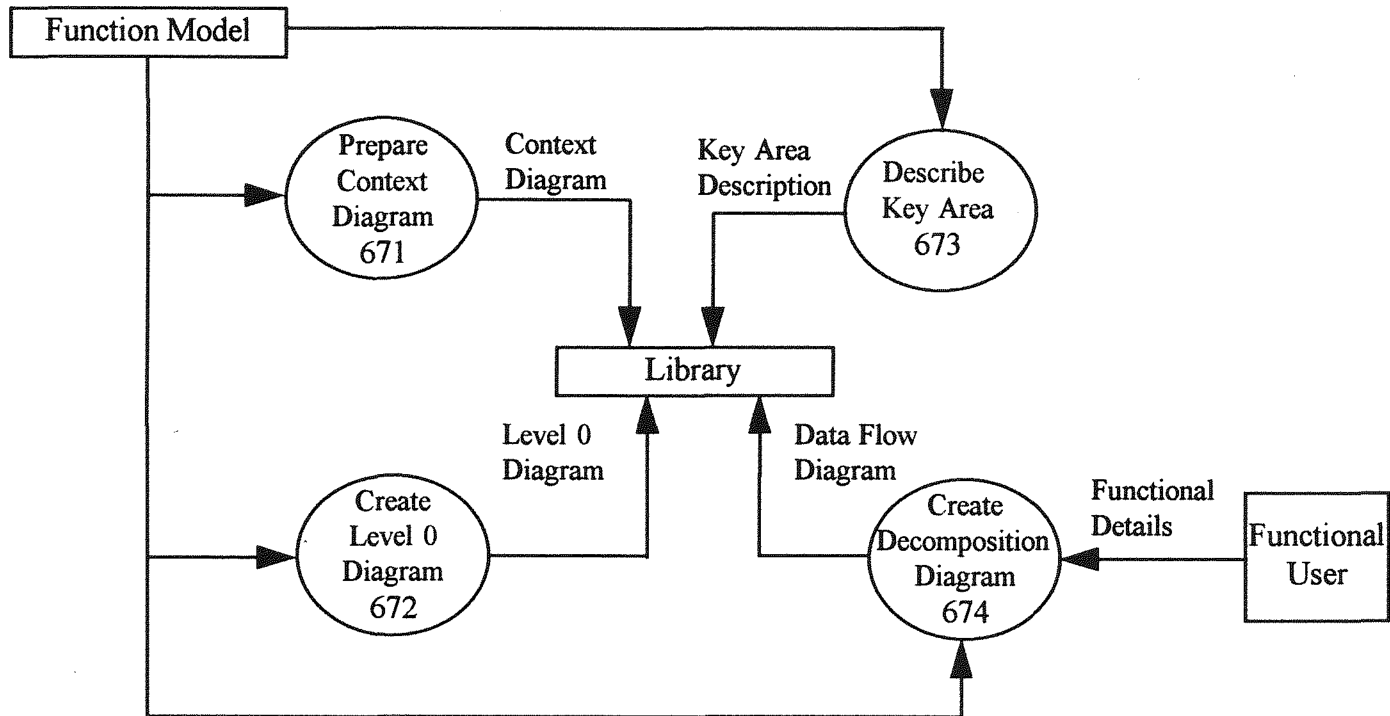
6.6.1 Distribute Draft Function Model. The draft function model is distributed to functional and technical analysts in the organization who did not participate in the reverse engineering project, as well as to the original members of the domain modeling team.

6.6.2 Collect Model Change. Comments and proposed model changes, additions, and deletions are collected and reviewed for clarity, validity, and justification.

6.6.3 Organize Model Change. Proposed model changes are organized by model section. Duplicates are consolidated into a single change package.

6.6.4 Resolve Model Discrepancy. Model discrepancies are resolved by reassembling the members of the domain analysis group (key area 4). The domain analysis modeling group reviews the structure and narrative content of the function model in facilitated modeling sessions. Discrepancies are identified and resolved and proposed changes are accepted or rejected.

6.6.5 Produce Function Model. The final function model is produced by making changes approved by the domain analysis modeling group.



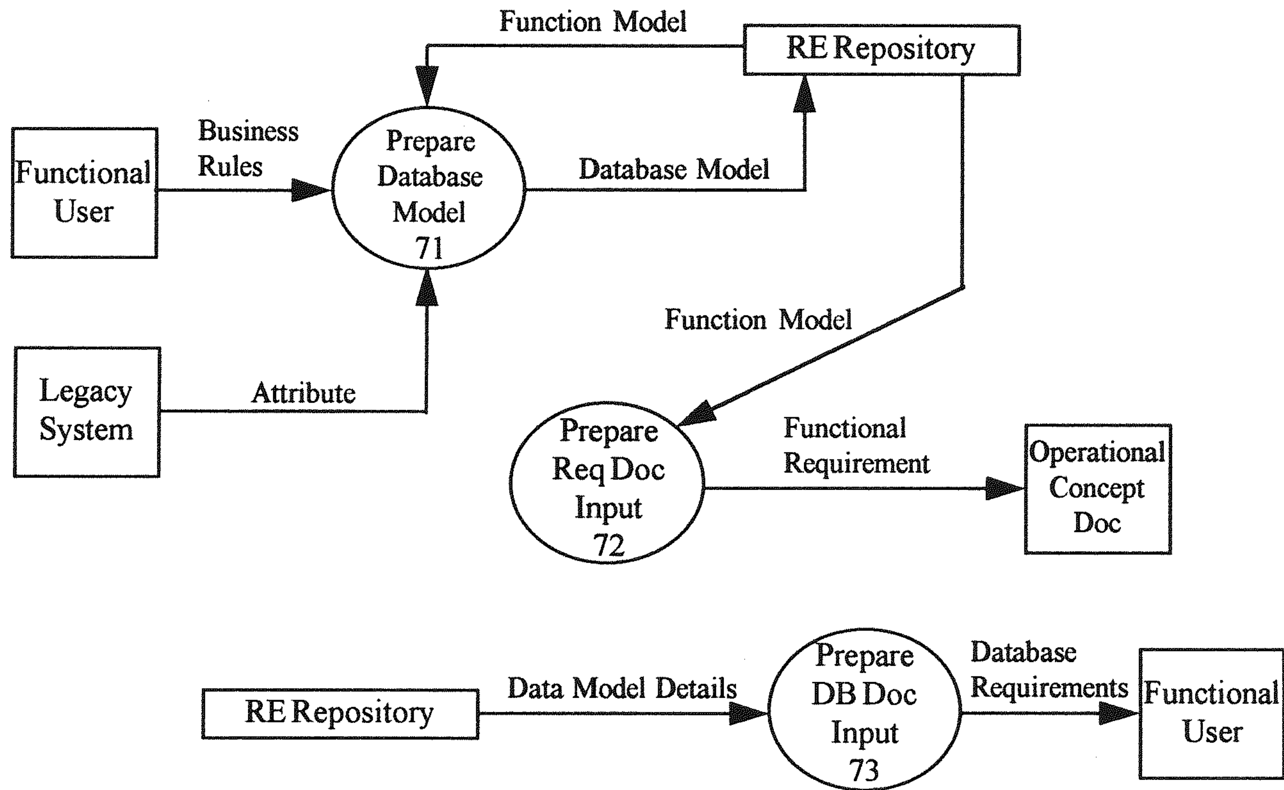
DFD 6.7 - Prepare visual process model

6.7.1 Prepare Context Diagram. A context diagram for the proposed system is prepared. The major external entities that provide data to and accept data from the system are represented by a single process.

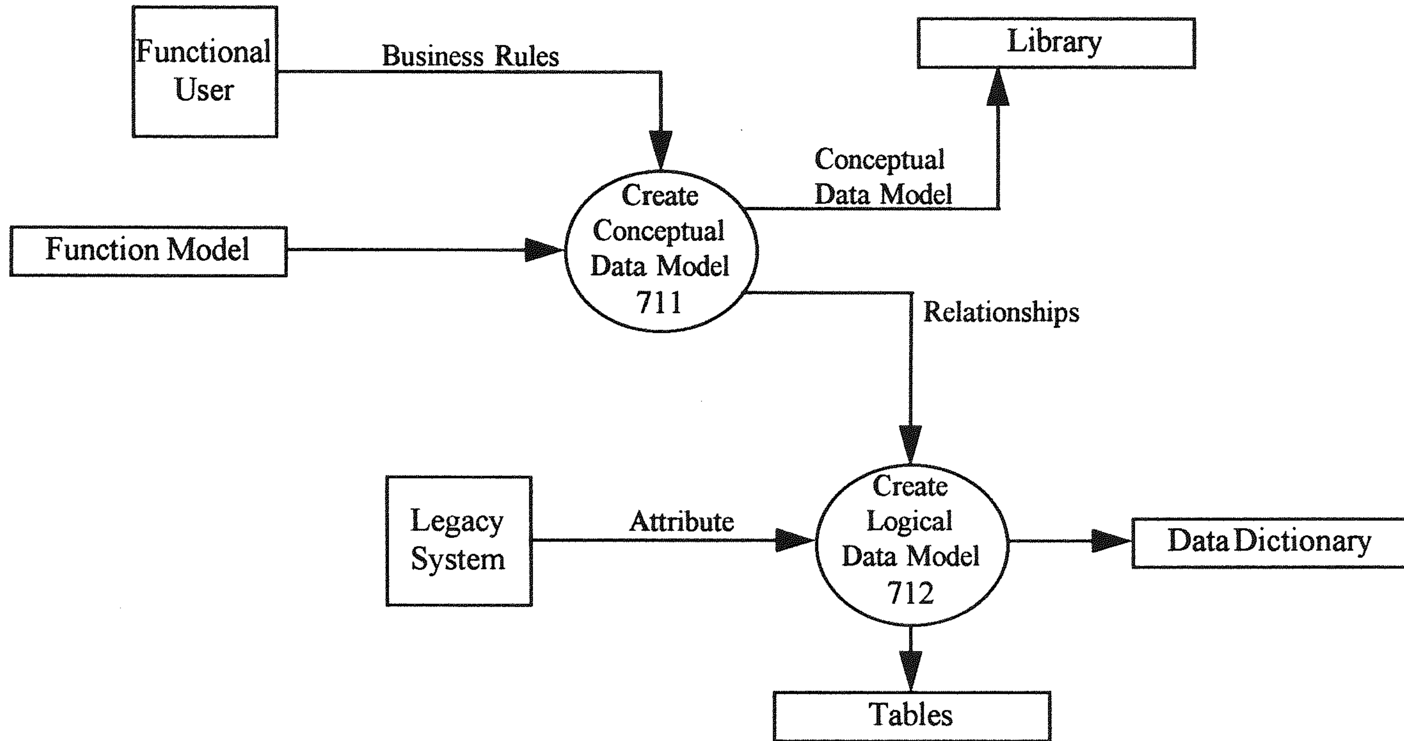
6.7.2 Create Level 0 Diagram. A level 0 diagram is the first lower-level decomposition of the proposed system and identifies the key areas identified in the function model. The level 0 diagram shows high-level system inputs and outputs, as well as interactions between the key areas.

6.7.3 Describe Key Area. Key areas on the level 0 diagram are described in a single paragraph to provide a high-level description of the functions performed. This is the single exception to the rule that only primitive-level activities are described.

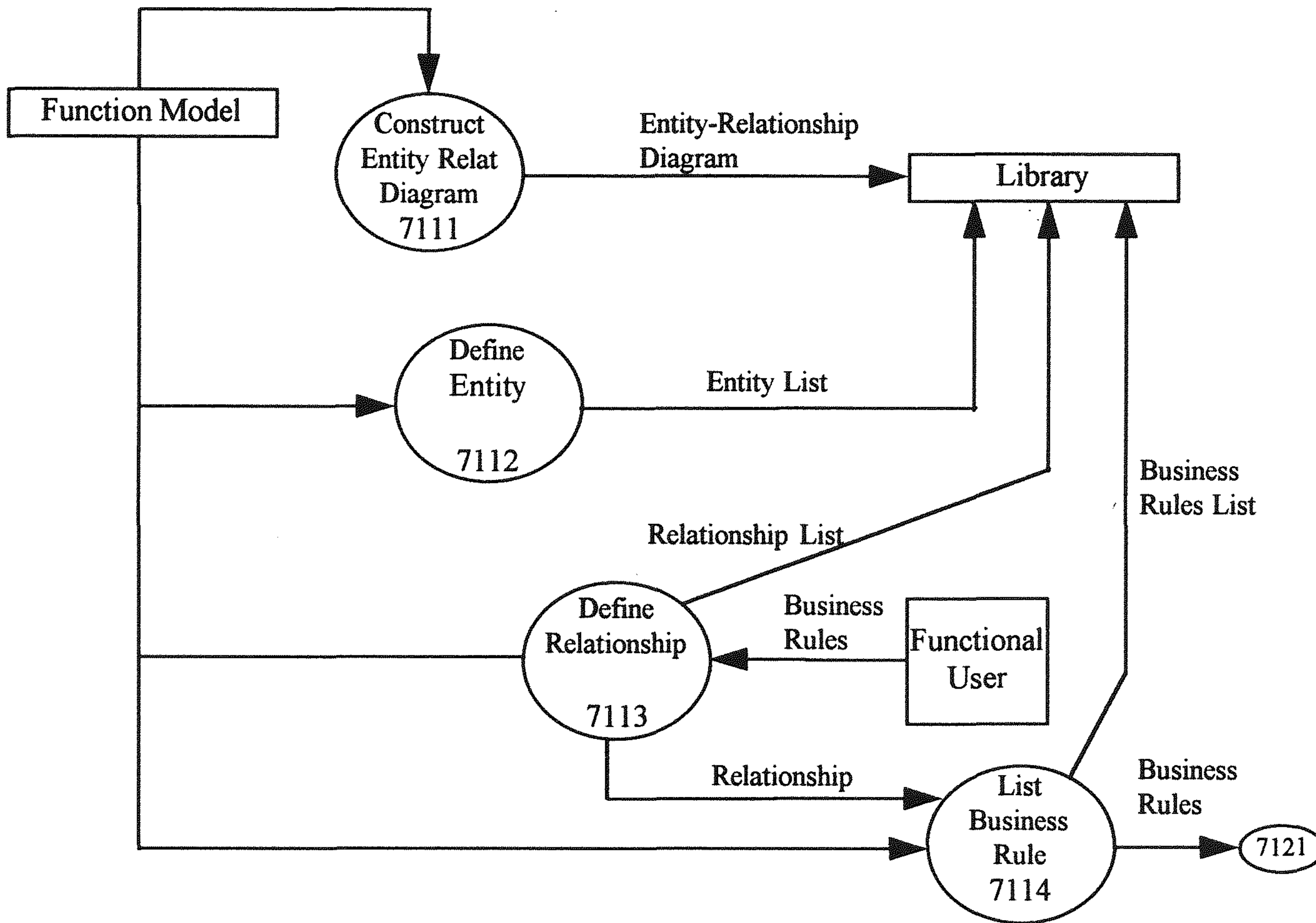
6.7.4 Create Decomposition Diagram. Lower-level decomposition diagrams are created by preparing a data flow diagram for each level in the final function model.



DFD 7 - Document design information



DFD 7.1 - Prepare database model



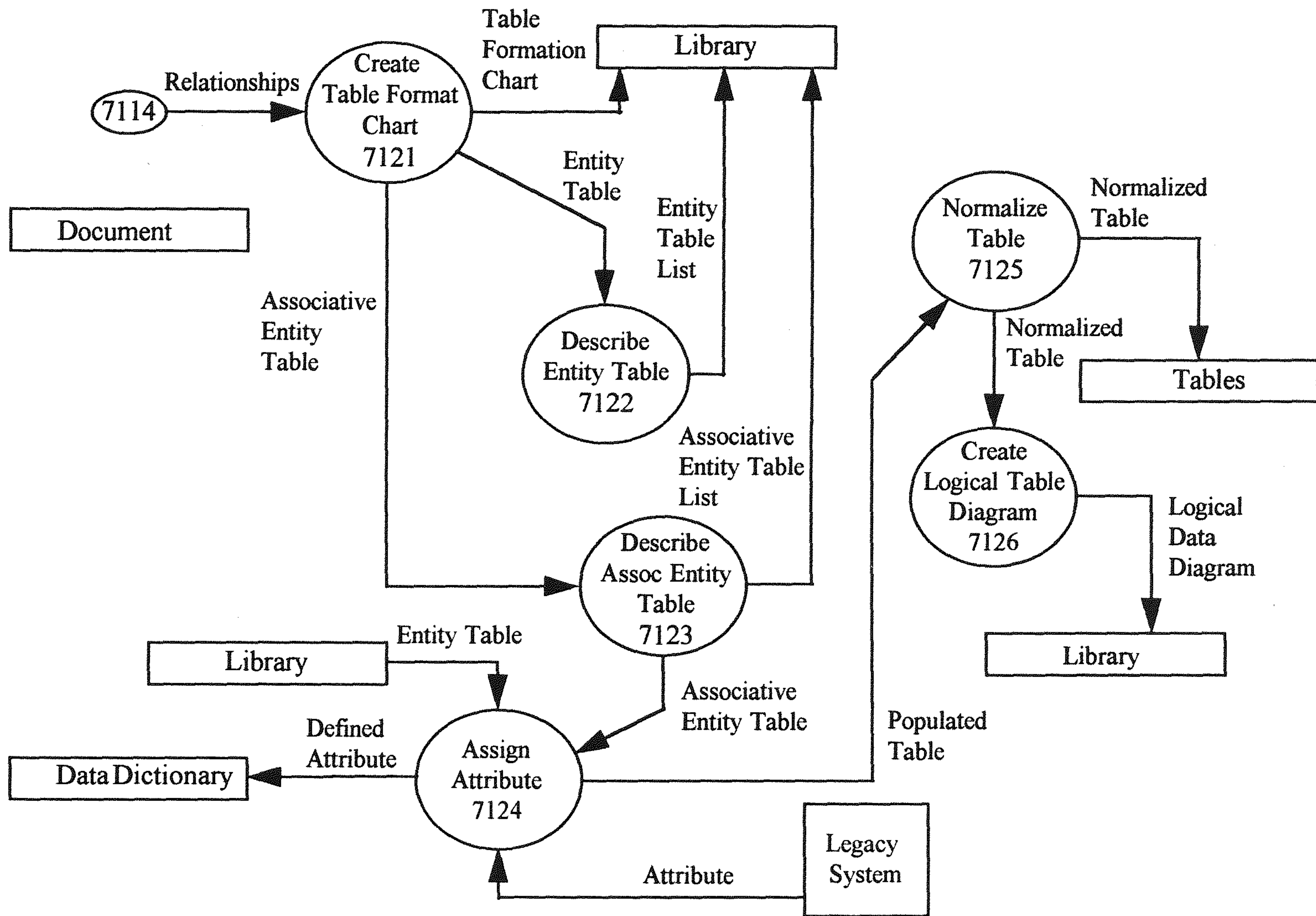
DFD 7.1.1 - Create conceptual data model

7.1.1.1 Construct Entity-Relationship Diagram (ERD). An ERD showing the conceptual data structure required to support the activities in the function model is constructed by identifying the entities and relationships that exist between them.

7.1.1.2 Define Entity. Each entity on the ERD is assigned a sequential number which identifies an entry on an entity definition list. Entities are defined with the assistance of functional users and domain specialists. The identifier (key) for each entity is also identified.

7.1.1.3 Define Relationship. Each relationship on the ERD is assigned a sequential number which identifies an entry in a relationship definition list. Relationships are defined with the assistance of functional users and domain specialists. The associated entities and their keys are also identified.

7.1.1.4 List Business Rule. Business rules for each relationship on the ERD are listed in clear narrative (the ERD is coded to show the same information). Business rules consist of two components: membership class and membership degree. Membership class represents obligatory or non-obligatory participation in a relationship. Membership degree represents the number of entity occurrences in a relationship (i.e., one to one, one to many, or many to many).



DFD 7.1.2 - Create logical data model

7.1.2.1 Create Table Formation Chart. A chart cross-referencing entities and relationships with the tables formed is created to allow objects on the ERD to be correlated with tables on the logical diagram. Tables are formed according to the membership class and membership degree of each relationship.

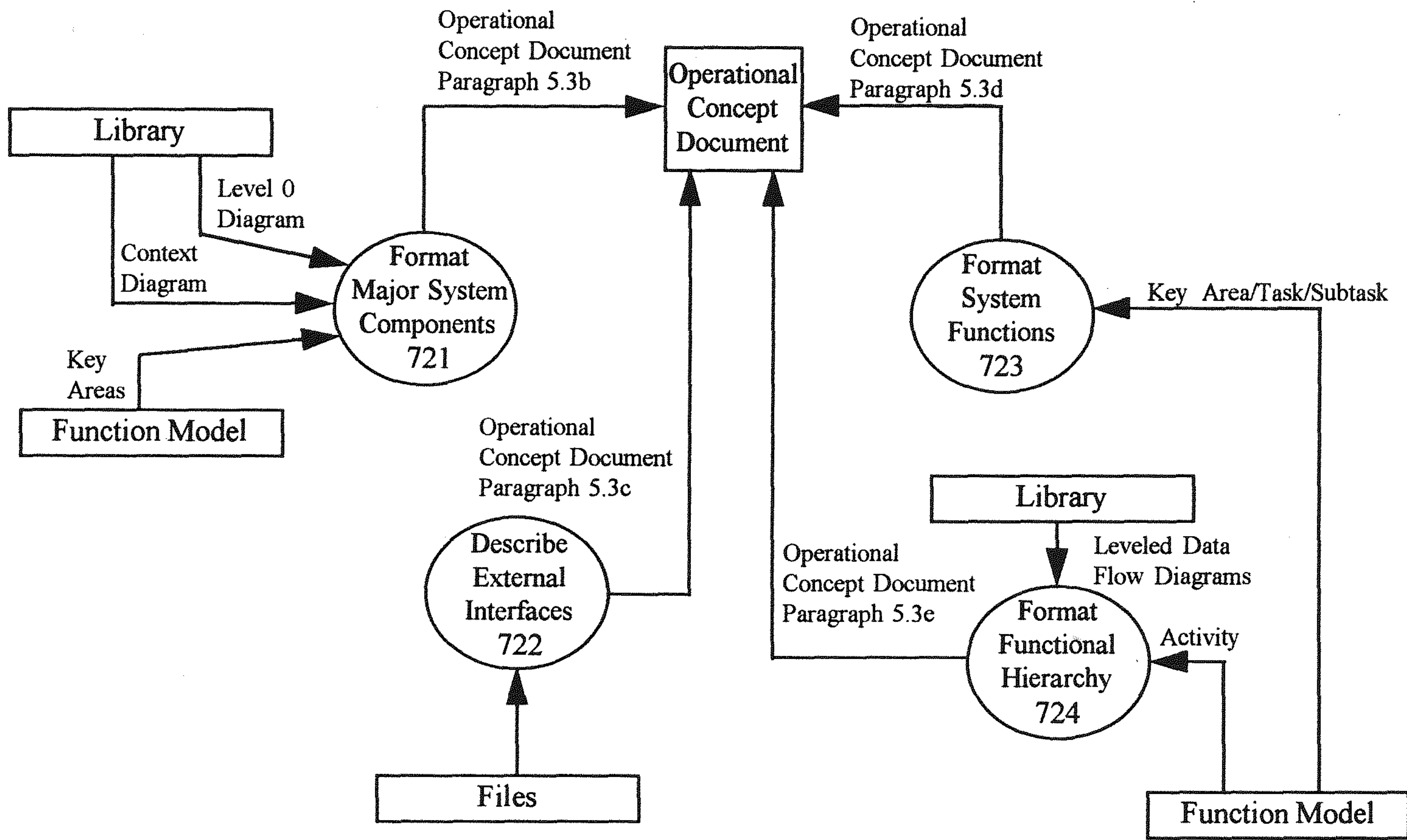
7.1.2.2 Describe Entity Table. Each entity on the ERD results in the formation of a logical table (a possible exception is a one-to-one relationship where the two entities are consolidated into a single table). The table is described and its primary key and foreign keys (if any) are identified.

7.1.2.3 Describe Associate Entity Table. Associate entity tables (join tables or relationship tables) are always created when there is a many-to-many relationship between two entities; they may be created under other circumstances. The table is described and its foreign keys (the primary keys from the two related tables) are identified.

7.1.2.4 Assign Attribute. Attributes (data elements) identified from legacy system files, databases, reports, and forms are assigned to a logical table.

7.1.2.5 Normalize Table. Tables are reduced to first normal form, second normal form, and then to third normal form to remove possible insertion, deletion, and update anomalies from the logical design. Additional tables may be generated in this activity.

7.1.2.6 Create Logical Data Diagram. A logical data diagram is created by representing each table as a rectangle on a chart. Table identifiers and foreign keys are also shown for each table. Tables are connected by drawing directed lines from primary keys to foreign keys. The logical table diagram represents the navigational paths between tables.



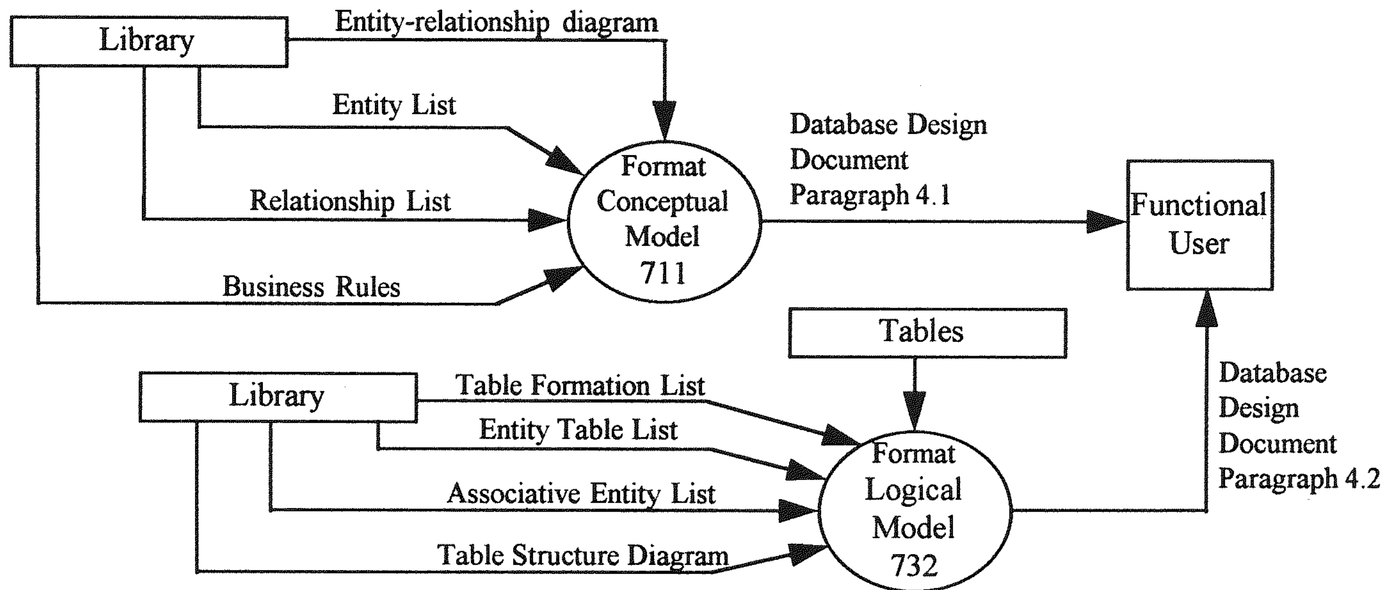
DFD 7.2 - Prepare requirements document input

7.2.1 Format Major System Components. The data flow context diagram, level 0 diagram, and function model key area descriptions are formatted for inclusion as Operational Concept Document paragraph 5.3.b.

7.2.2 Describe External Interface. External interfaces identified during the reverse engineering effort are described in summary form to satisfy the requirements for Operational Concept Document paragraph 5.3.c.

7.2.3 Format System Function. Function model key areas, tasks, and subtasks are formatted for inclusion in Operational Concept Document paragraph 5.3.d.

7.2.4 Format Functional Hierarchy. The function model activity descriptions and associated data flow diagrams are formatted for inclusion in Operational Concept Document paragraph 5.3.e. These two model components satisfy the requirement to show charts and descriptions describing inputs, outputs, data flow, and manual and automated processes.



DFD 7.3 - Prepare database document input

7.3.1 Format Conceptual Model. The ERD, entity and relationship definitions, and business rules list are formatted for inclusion in the Database Design Description paragraph 4.1. Individual documents are identified as subparagraphs.

7.3.2 Format Logical Model. The table formation chart, table description, logical data diagram, and table layouts are formatted for inclusion in Database Design Description paragraph 4.2. Individual documents are identified as subparagraphs.

Appendix C

Reverse Engineering Methodology
Logical Data Model
Table Descriptions

Table Name: Acronym
Table Number: 1

Key: No-Seq-Acronym (counter)
Foreign Key: None

Attributes:	Type	Size
Acronym	C	15
Name-Long	C	50
Description-Acronym	C	255
Reference	C	50
Date-Description	Date	-
Author-Name	C	20

Table Name: Attribute
Table Number: 2

Key: No-Seq-Attribute (counter)
Foreign Key: None

Attributes:	Type	Size
Name-Attribute	C	50
Type-Attribute	C	2
Size-Attribute	N	2
Description-Attribute	C	255

Table Name: Comment
Table Number 3

Key: No-Seq-Comment (counter)
Foreign Key: None

Attributes:	Type	Size
No-Line-Begin	N	6
Text-Comment	C	255
Type-Comment	C	1
Author-Comment	C	20
Date-Comment	Date	-

Table Name: Component
Table Number: 4

Key: ID-Component (C, 8)
Foreign Key: No-Seq-Metric

Attributes:	Type	Size
Type-Prog	C	2
Class-Prog	C	2
Rating-Structure	N	2
Rating-Comments	N	2
Rating-Names	N	2
Index-Complexity	N	2
Name-Func-Tech	C	20
Name-Domain-Spec	C	20
Name-RE	C	20
No-Files-In	N	2
No-Files-Out	N	2
No-Files-IO	N	2
No-Screens	N	2
No-Reports	N	2
Language	C	10
No-Group	N	3
No-Lines-Source	N	6
Name-Short	C	50
Date-Written	Date	-
No-Versions	N	3
No-Authors	N	3
CICS-Trans-ID	C	4
No-Data-Div-Lines	N	6
No-Proc-Div-Lines	N	6
No-Var-Work-Stor	N	6
No-Prog-Called	N	2
No-GOTO-Stmnts	N	3
No-Perf-Stmnts	N	6
No-Paragraphs	N	6
No-Para-Lines-Avg	N	6
No-Redefine-Stmnts	N	6
Percent-Complete	N	3
Initial-Review-Complete	C	1
Purpose-Text	C	255
Objectives-Text	C	255
Assumptions-Text	C	255
Constraints-Text	C	255
Avg-Data-Name-Length	N	6

Table Name: Component-Comment
Table Number 5

Foreign Key: ID-Component
Foreign Key: No-Seq-Comment

Attributes:	Type	Size
None		

Table Name: Component-Component
Table Number 6

Foreign Key: ID-Component-Calls
Foreign Key ID-Component-Called

Attributes:	Type	Size
Data-Pass	Logical	1
List-Parms	C	50

Table Name: Component-Document
Table Number: 7

Foreign Key: ID-Component
Foreign Key: No-Seq-Doc

Attributes:	Type	Size
None		

Table Name: Component-File
Table Number: 8

Foreign Key: ID-Component
Foreign Key: ID-File

Attributes:	Type	Size
Name-File-Internal	C	32
Name-File-External	C	15
Organization-File	C	20
Access-File	C	20
Activity-File	C	2

Table Name: Component-Paragraph
Table Number: 9

Foreign Key: ID-Component
Foreign Key: No-Seq-Para

Attributes:	Type	Size
None		

Table Name: Component-Screen
Table Number: 10

Foreign Key: ID-Component
Foreign Key: ID-Screen

Attributes:	Type	Size
Activity-Screen	C	10

Table Name: Component-Table
Table Number: 11

Foreign Key: ID-Component
Foreign Key: No-Seq-Table

Attributes:	Type	Size
Action-Table	C	1

Table Name: Component-Transaction
Table Number: 12

Foreign Key: ID-Component
Foreign Key: Doc-ID-Code

Attributes:	Type	Size
Doc-Activity	C	1

Table Name: Document
Table Number: 13

Key: No-Seq-Doc (counter)
Foreign Key: None

Attributes:	Type	Size
Type-Doc	C	5
No-Index	C	10
Evaluation-Text	C	50
Comments	C	255
Doc-Name	C	50
Requiring-Directiv	C	15
No-Pages	N	5
Location	C	20
Date-Written	Date	-
Date-Last-Change	Date	-
Usefulness-Rating	N	2

Table Name: Element
Table Number 14

Key: No-Seq-Element (Counter)
Foreign Key: None

Attributes:	Type	Size
Name-Element	C	32
Type-Element	C	1
Picture-Element	C	15
Layout-Structure	C	255
Description	C	255

Table Name: File
Table Number: 15

Key: ID-File (C, 10)
Foreign Key: None

Attributes:	Type	Size
Name-File-Short	C	50
Type-File	C	1
Media	C	10
Description-File	C	255

Table Name: File-Record
Table Number: 16

Foreign Key: ID-File
Foreign Key: No-Seq-Record

Attributes:	Type	Size
None		

Table Name: Function
Table Number: 17

Key: No-Seq-Function (counter)
Foreign Key: None

Attributes:	Type	Size
Text-Function	C	Memo
Author-Text	C	20
Date-Written	Date	-

Table Name: Metric
Table Number 18

Key: No-Seq-Metric (counter)
Foreign Key: None

Attributes:	Type	Size
Time-Analysis-Initial	N	6
Time-Analysis-Revised	N	6
Time-Analysis-Final	N	6
Time-Analysis-Actual	N	6
Time-Review-Initial	N	6

Table Name: Narrative
Table Number: 19

Key: No-Seq-Narrative (counter)
Foreign Key: None

Attributes:	Type	Size
Text-Narrative	C	Memo

Table Name: Narrative-Function
Table Number: 20

Foreign Key: No-Seq-Narrative
Foreign Key: No-Seq-Function

Attributes:	Type	Size
None		

Table Name: Note
Table Number: 21

Key: No-Seq-Note (counter)
Foreign Key: None

Attributes:	Type	Size
Text-Note	C	255
No-Line-Begin	N	6
Type-Note	C	1

Table Name: Object
Table Numbr: 22

Key: No-Seq-Object (counter)
Foreign Key: None

Attributes:	Type	Size
Name-Object	C	20
Type-Object	C	20
Description-Object	C	255

Table Name: Paragraph
Table Number: 23

Key: No-Seq-Para (counter)
Foreign Key: None

Attributes:	Type	Size
Name-Para	C	32
No-Para-Line	N	6
Group-No	N	3

Table Name: Paragraph-Function
Table Number: 24

Foreign Key: No-Seq-Para
Foreign Key: No-Seq-Function

Attributes:	Type	Size
None		

Table Name: Paragraph-Note
Table Number: 25

Foreign Key: No-Seq-Para
Foreign Key: No-Seq-Note

Attributes:	Type	Size
None		

Table Name: Process
Table Number: 26

Key: No-Seq-Process (counter)
Foreign Key: None

Attributes:	Type	Size
Process-Text	C	255
No-Level	N	1
Type-Para	C	2
Model-Para-No	N	8
Title-Process	C	255
Comment	C	15

Table Name: Process-Function
Table Number: 27

Foreign Key: No-Seq-Process
Foreign Key: No-Seq-Function

Attributes:	Type	Size
None		

Table Name: Process-Process
Table Number: 28

Foreign Key: Is-Parent-Of
Foreign Key: Is-Child-Of

Attributes:	Type	Size
None		

Table Name:	Record		
Table Number:	29		

Key:	No-Seq-Record	(counter)	
Foreign Key:	Doc-ID-Code		

Attributes:	Type	Size
Record-Name	C	30
Description-Record	C	255

Table Name:	Record-Element
Table Number:	30

Foreign Key:	No-Seq-Record
Foreign Key:	No-Seq-Element

Attributes:	Type	Size
None		

Table Name:	Screen
Table Number:	31

Key:	ID-Screen	(C, 10)
Foreign Key:	None	

Attributes:	Type	Size
Name-Screen	C	15
Screen-Type	C	1

Table Name: Table
Table Number: 32

Key: No-Seq-Table (counter)
Foreign Key: None

Attributes:	Type	Size
Name-Table	C	15
Table-Description	C	255
Data-View-Name	C	8

Table Name: Table-Attribute
Table-Number: 33

Foreign Key: No-Seq-Table
Foreign Key: No-Seq-Attribute

Attributes:	Type	Size
None		

Table Name: Term
Table-Number: 34

Key: No-Seq-Term (counter)
Foreign Key: None

Attributes:	Type	Size
Term	C	15
Description-Term	C	255
Reference	C	25
Date-Description	Date	-
Author-Name	C	25

Table Name: Transaction
Table-Number: 35

Key: Doc-ID-Code (C, 3)
Foreign Key: None

Attributes:	Type	Size
Description-DIC	C	255

Appendix D

Permission to Use Source Code

DEPARTMENT OF THE AIR FORCE
MATERIEL SYSTEMS GROUP (AFMC)

6 FEB 1995

MEMORANDUM FROM AFMC MSG/SH
4225 Logistics Ave., Ste 22
Wright-Patterson AFB OH 45433-5761

FOR: MR. ROBERT L. MILLER
3243 Windmill Dr.
Beavercreek OH 45432

SUBJECT: Request for Government Software and Access to Documentation

1. We have received a response from our legal office about the information you requested in your letter to 88 ABW/JAC, 14 December 1994. Although they had no legal objection to granting you access to use the data for your dissertation (however, you must sign a Memorandum of Agreement), we are having a problem identifying exactly what portion of the D035 system you need. CPCI 1 of the D035 system is the Item Manager Wholesale Requisition System (D035A). There is no subsystem of D035A called "Cataloging Management Control Data Subsystem".

2. Since there is some confusion as to exactly what subsystem code is needed, I would suggest that you contact my OPR for the D035 system, Ms. Laurie Wohlers, at 257-1500 extension 3402 to clarify your requirement. At that time we will be happy to work with you to provide the code and documentation.

BRIAN F. DREW
Director
Asset Management DSM

Appendix E

Biographical Sketch of Student

Biographical Sketch of Student

Robert Miller is a senior project manager with I-NET, Incorporated, Dayton, Ohio. Currently he is supporting the U.S. Air Force by investigating methods for making legacy system information more readily available to users.

He was born in Sidney, Ohio, in 1941. After enlisting in the US Army in 1959, he served as an administrative specialist. Assignments in this field took him to Okinawa and Fort Sill, Oklahoma. In 1966 he attended the Department of State's Foreign Service Institute intensive Japanese course. After a tour of duty in Hawaii, in 1969 he was selected as a data processing student in the Army's civil schooling program. He earned an AA in Business Information Systems from Orange Coast College in 1972.

After serving as a machine room shift supervisor with a field data processing unit in the Republic of Vietnam, he was assigned to the Pentagon, Washington, DC. As a part-time student over the next six years, he completed a BS in Technology of Management from The American University and a MS in Systems Management from the University of Southern California.

He was selected to attend the Army's Sergeants Major Academy in 1978 and upon graduation was assigned to the military detachment of the Pacific Stars and Stripes newspaper in Tokyo, Japan. While stationed there he was able to fulfill his ambition to combine his Japanese and data processing training by working part-time for a Japanese software company. Following a one-year assignment in Fort Huachuca, Arizona, he retired from the Army as a Sergeant Major in 1983.

Returning to Dayton, Ohio, he was employed by Contel Information Systems and later Century Technologies as a specialist in structured techniques and information engineering. He joined I-NET in 1996.

He decided to pursue a doctoral degree in 1992, and after examining several programs selected Nova Southeastern University because the school offered a degree in an area of interest to him and because the program was designed for the working student.

Completion of the Doctor of Philosophy degree in Information Systems will enhance his knowledge and appreciation for technology, planning, and management within the information technology field.

References

Abelson, H., & Sussman, G. J. (1985). *Structure and interpretation of computer programs.* New York: McGraw-Hill.

Aiken, P. H. (1996). *Data reverse engineering: Slaying the legacy dragon.* New York: McGraw-Hill.

Al-Jarrah, M. M., & Torsun, I. S. (1979). An empirical analysis of COBOL programs. *Software: Practice and Experience, 9,* 341-359.

Ambras, J., & O'Day V. (1988). Microscope: A knowledge-based programming environment. *IEEE Software, 5,* 50-58.

ANSI/IEEE Standard 610.12-1990. (1990). *IEEE Standard Glossary of Software Engineering Terminology.* New York: IEEE.

Antonini, P., Benedusi, P., Cantone, G., & Cimitile, A. (1987). Maintenance and reverse engineering: Low-level design documents production and improvement. *Proceedings of the IEEE Conference on Software Maintenance CSM '87*, 91-100.

Arango, G., Baxter, I., & Freeman, P. (1985). Maintenance and porting of software by design recovery. *Proceedings of the IEEE Conference on Software Maintenance CSM '85*, 42-49.

Arango, G., Baxter, I., Freeman, P., & Pidgeon, C. (1986). TMM: Software maintenance by transformation. *IEEE Software, 3,* 27-39.

Arango, G., & Prieto-Díaz, R. (1991). Domain analysis concepts and research directions. In G. Arango & R. Prieto-Díaz (Eds.), *IEEE tutorial on domain analysis and software systems modeling* (pp. 9-32). Los Alamitos, CA: IEEE Computer Society Press.

Atkins, M. (1994, September). Building a better bridge. *Software Magazine, 14*, 6,8.

Avellis, G., Iacobbe, A., Palmisano, D., Semeraro, G., & Tinelli, C. (1991). An analysis of incremental assistant capabilities of a software evolution expert system. *Proceedings of the IEEE Conference on Software Maintenance CSM '91*, 220-227.

Bachman, C. (1988, July 1). A CASE for reverse engineering. *Datamation, 34,* 49-56.

Baker, M. J., & Eick, S. G. (1994). Visualizing software systems. *Proceedings of the 16th International IEEE Conference on Software Engineering*, 59-67.

Baxter, I. D. (1992). Design maintenance systems. *Communications of the ACM, 35,* 73-89.

Beck, J. & Eichmann, D. (1993). Program and interface slicing for reverse engineering. *Proceedings of the 15th IEEE International Conference on Software Engineering,* 509-518.

Benedusi, P., Cimitile, A., & De Carlini, U. (1989). A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance. *Proceedings of the IEEE Conference on Software Maintenance CSM '89*, 180-189.

Benedusi, P., Cimitile, A. & de Carlini, U. (1992). Reverse engineering processes, design document production, and structure charts. *Journal of Systems Software, 19,* 225-245.

Bennett, K. H. (1981). Automated support of software maintenance. *Information and Software Technology, 33,* 74-85.

Bennett, K. H. (1993). An overview of maintenance and reverse engineering. In H. J. van Zuylen (Ed.), *The REDO compendium: Reverse engineering for software maintenance* (pp. 13-34). Chichester, England: John Wiley & Sons.

Berns, G. (1984). Assessing software maintainability. *Communications of the ACM, 27,* 14-23.

Biggerstaff, T. J. (1989). Design recovery for maintenance and reuse. *IEEE Computer, 22,* 36-49.

Biggerstaff, T. J., Mitbander, B. G., & Webster, D. (1994). The concept assignment problem in program understanding. *Proceedings of the 15th IEEE Conference on Software Engineering*, 482-498.

Boehm, B. W. (1981). *Software engineering economics*. Englewood Cliffs, NJ: Prentice-Hall.

Boehm, B. W. (1987). Improving software productivity. *IEEE Computer, 20,* 43-57.

Boehm-Davis, D. A., Holt, R. W., & Schultz, A. C. (1992). The role of program structure in software maintenance. *International Journal of Man-Machine Studies, 36,* 21-63.

Breuer, P. T., & Lano, K. C. (1991). Creating specifications from code: Reverse engineering techniques. *Journal of Software Maintenance: Research and Practice, 3,* 145-162.

Britcher, R. N., & Craig, J. J. (1986). Using modern design practices to upgrade aging software systems. *IEEE Software, 3,* 16-24.

Brown, A. J. (1993). Specifications and reverse-engineering. *Software Maintenance: Research and Practice, 5,* 147-153.

Brown, P. (1983). Why does software die? In R. S. Arnold (Ed.), *IEEE tutorial on software restructuring* (pp. 109-116). Washington, DC: IEEE Computer Society Press.

Buckley, W. (1972). A systems approach to epistemology. In G. J. Klir (Ed.), *Trends in general systems theory* (pp. 188-202). New York: John Wiley & Sons.

Burson, S., Kotik, G., & Markosian, L. (1990). A program transformation approach to automating software re-engineering. *Proceedings of the 14th Annual International Computer Software and Application Conference COMPSAC-90,* 314-322.

Bush, E. (1993, April). Flying car now surfs. *Software Magazine, 13,* 6.

Byrne, E. J. (1991). Software reverse engineering: A case study. *Software--Practice and Experience, 21,* 1349-1364.

Calliss, F. W., Khalil, M., Munro, M., & Ward, M. (1988). A knowledge-based system for software maintenance. *Proceedings of the IEEE Conference on Software Maintenance CSA '88,* 319-324.

Canfora, G., Cimitile, A., & De Carlini, U. (1992). A logic-based approach to reverse engineering tools production. *IEEE Transactions on Software Engineering, 18,* 1053-1064.

Canfora, G., Cimitile, A., & Munro, M. (1994). RE^2: Reverse engineering and reuse re-engineering. *Software Maintenance: Research and Practice, 6,* 53-72.

Canfora, G., Sansone, L., & Visaggio, G. (1992). Data flow diagrams: Reverse engineering production and animation. *Proceedings of the IEEE Conference on Software Maintenance CSM '92,* 366-375.

Cezzar, R. (1989). *The essentials of COBOL I.* Piscataway, NJ: Research and Education Association.

Chen, S., Heisler, K. G., Tsai, W. T., Chen, X., & Leung, E. (1990). A model for assembly program maintenance. *Journal of Software Maintenance: Research and Practice, 2,* 3-32.

Chikofsky, E. J., & Cross, J. H. (1990). Reverse engineering and design recovery, a taxonomy. *IEEE Software, 7,* 13-17.

Choi, E. M. (1993). Support for program understanding during maintenance via chunking. *Dissertation Abstracts International, 54,* 2060B. (University Microfilms No. DA9324219)

Choi, S. C., & Scacchi, W. (1990). Extracting and restructuring the design of large systems. *IEEE Software, 7,* 66-71.

Cleveland, L. (1989). A program understanding support environment. *IBM Systems Journal, 28,* 324-344.

Cohen, D. I. A. (1991). *Introduction to computer theory.* New York: John Wiley & Sons.

Collofello, J. S., & Blaylock, J. W. (1985). Syntactic information useful for software maintenance. *AFIPS Conference Proceedings, 54,* 547-553.

Connal, D. G., & Burns, D. R. (1993, October). Reverse engineering: Getting a grip on legacy systems. *Data Management Review, 3,* 24-27.

Corbi, T. A. (1989). Program understanding: Challenge for the 1990s. *IBM Systems Journal, 28,* 294-306.

Cross, J. H, Chikofsky, E. J., & May, C. H. (1992). Reverse engineering. *Advances in Computers, 35,* 199-254.

Cunningham, J. (1962). Why COBOL? *Communications of the ACM, 5,* 236-253.

Davenport, T. H., (1993). *Process Innovation: Reengineering Work Through Information Technology.* Boston: Harvard Business School.

Darlison, A. G., & Sabanis, N. (1993) Data remodeling. In H. J. van Zuylen (Ed.), *The REDO compendium: Reverse engineering for software maintenance* (pp. 311-325). Chichester, England: John Wiley & Sons.

Davis, A. M. (1988). A comparison of techniques for the specification of external system behavior. *Communications of the ACM, 31,* 1098-1115.

Davis, J. (1990, July/August). CASE environments and re-engineering. *CASE Trends, 2,* 1-6.

Davis, J. (1991a, Summer). Software re-engineering: A beginner's guide. *CASE Trends, 3,* 10-16.

Davis, J. (1991b, Fall). Software re-engineering: Capture tools. *CASE Trends, 3,* 30, 33-34.

Davis, R. K., & Shah, A. D. (1985). Service analysis: Key to effective performance management. *Journal of Capacity Management, 3,* 1-21.

Debaud, J., Moopen, B., & Rugabers, S. (1994). Domain analysis and reverse engineering. *Proceedings of the 1994 IEEE International Conference on Software Maintenance CSM '94,* 326-335.

Debest, X. A., Rüdiger, K., & Wagner, J. (1992). REVENG: A cost-effective approach to reverse engineering. *ACM SIGSOFT Software Engineering Notes, 17,* 60-67.

Desmond, J. (1992, May). Reengineering reality check [Editor's letter]. *Software Magazine, 12,* 1.

Devanbu, P., Brachman, R. J., Selfridge, P. G., & Ballard, B. W. (1991). LaSSIE: A knowledge-based software information system. *Communications of the ACM, 34,* 35-49.

Dietrich, S. W. & Callis, F. W. (1992). A conceptual design for a code analysis knowledge base. *Journal of Software Maintenance: Research and Practice, 4,* 19-36.

Dock, V. T. (1979). *Structured COBOL: American national standard.* St. Paul, MN: West.

Edwards, H. M., & Munro, M. (1993). RECAST: Reverse engineering from COBOL to SSADM specifications. *Proceedings of the 15th IEEE International Conference on Software Engineering,* 499-508.

Eliot, L. B. (1992, July). Software review: COBOL Analyst, SEEC, Inc. *CASE Trends, 4,* 68-70.

Elshoff, J. L., & Marcotty M. (1982). Improving computer program readability to aid modification. *Communications of the ACM, 25,* 512-521.

Fairley, R. E. (1985). *Software engineering concepts.* NY: McGraw-Hill.

Fiorello, M., & Cugini, J. (1984). Is COBOL-8x cost effective? *AFIPS Conference Proceedings, 53,* 223-228.

FIPS PUB 106 (1984, June 15). Federal Information Processing Standard 106. Guideline on Software Maintenance. Chapter 5. System Maintenance vs. System Redesign, pp. 14-17. Department of Commerce, National Institute of Standards and Technology.

Friedlander, P., & Toothman, W. E. (1994). Reengineering done right: Intermediate solutions that are cost effective. *Information Systems Management, 11,* 7-15.

Friedman, A. L., & Cornford, D. S. (1989). *Computer systems development: History, organization and implementation.* New York: John Wiley & Sons.

Garnett, E. S. & Mariani, J. A. (1990). Software reclamation. *Software Engineering Journal, 5,* 185-191.

Gelertner, D., & Jagannathan, S. (1990). *Programming Linguistics.* Cambridge, MA: MIT Press.

Gillis, K. D. & Wright, D. G. (1990). Improving software maintenance using system-level reverse engineering. In *Proceedings of the IEEE Conference on Software Maintenance CSM '90*, 84-90. Washington, DC: IEEE Computer Society Press.

Goguen, N. H. (1975). Control structures for structured programming in COBOL. In *Proceedings of a Symposium on Structured Programming in COBOL--Future and Present* (pp. 68-87). New York: Association for Computing Machinery.

Gopal, R., & Schach, S. R. (1989). Using automatic program decomposition techniques in software maintenance tools. *Proceedings of the IEEE Conference on Software Maintenance CSM '89*, 132-141.

Griswold, W. G., & Notkin, D. (1992). Computer-aided vs. manual program restructuring. *ACM SIGSOFT Software Engineering Notes, 17,* 33-41.

Grumann, J., & Welch, P. J. (1992). A graph method for technical documentation and re-engineering of DP applications. In P. A. V. Hall (Ed.), *Software reuse and reverse engineering in practice* (pp. 321-353). London: Chapman & Hall.

Hall, P. A. V. (1992). Software reuse, reverse engineering and re-engineering. In P. A. V. Hall (Ed.), *Software Reuse and Reverse Engineering in Practice* (pp. 3-31). London: Chapman and Hall.

Handel, Y., & Degtyar, B. (1994). *The revolutionary guide to COBOL.* Birmingham, United Kingdom: WROX Press.

Hanna, M. (1993, July). Can CASE bridge to object world? *Software Magazine, 13,* 41-45.

Harandi, M. T., & Ning, J. Q. (1988). PAT: A knowledge-based program-analysis tool. In *Proceedings of the Conference on Software Maintenance* (pp. 312-318). Washington, DC: IEEE Computer Society Press.

Harandi, M. T., & Ning, J. Q. (1990). Knowledge-based program analysis. *IEEE Software*, *7,* 74-81.

Harrold, M. J., & Malloy B. (1993). A unified interprocedural program representation for a maintenance environment. *IEEE Transactions on Software Engineering, 19,* 584-593.

Hausler, P. A., Pleszkoch, M. G., Linger, R. C., & Hevner, A. R. (1990). Using function abstraction to understand program behavior. *IEEE Software, 7*(1), 55-63.

Hayes, I. S. (1993, October). Product review: Legacy Workbench, KnowledgeWare, Inc. *Data Management Review, 3,* 40.

Hayes, I. S. (1994, September). Protect software assets by migrating legacies. *Application Development Trends, 1,* 65-66, 68, 70-73.

Hayley, K., Plewa, J., & Watts, M. (1993, April 15). Reengineering tops CIO menu. *Datamation, 39,* 73-74.

Hennell, M. A., McNicol, W. M., & Hawkins, J. (1980). The static analysis of COBOL programs. *ACM SIGSOFT Software Engineering Notes, 5,* 17-23.

Hickey, G. L., & Jennings, R. A. (1994, February). USAA's reengineering turns IE on its head. *Application Development Trends, 1,* 18-21.

Hicks, J. R. (1975). Suggested changes to COBOL to facilitate structured programming. In H. P. Stevenson (Ed.), *Proceedings of a Symposium on Structured Programming in COBOL--Future and Present* (pp. 88-94). New York: Association for Computing Machinery.

Holloway, S. (1992). Re-engineering business systems to use the next generation of software. In P. A. V. Hall (Ed.), *Software reuse and reverse engineering in practice* (pp. 271-282). London: Chapman & Hall.

Howden, W. E., & Pak, S. (1992). Problem domain, structural and logical abstractions in reverse engineering. In *Proceedings of the IEEE Conference on Software Maintenance CSM '92 (*214-224). Los Alamitos, CA: IEEE Computer Society Press.

Howden, W. E., & Wieand, B. (1994). QDA--A method for systematic informal program analysis. *IEEE Transactions on Software Engineering, 20*, 445-462.

Howe, D. R. (1983). *Data analysis for data base design.* London: Edward Arnold.

Jacobson, I. & Lindström, F. (1991). Re-engineering of old systems to an object-oriented architecture. In *Proceedings of OOPSLA 1991* (pp. 340-350). New York: Association for Computing Machinery.

Johnson, W. L. & Soloway, E. (1985). PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering, 11,* 267-275.

Joiner, J. K., Tsai, W. T., Chen, X. P., Subramanian, S., Sun, J., & Gandamaneni, H. (1994). Data-centered program understanding. In *Proceedings of the 1994 IEEE International Conference on Software Maintenance CSM '94* (pp. 272-281). Los Alamitos, CA: IEEE Computer Society Press.

Kaposi, A. & Pyle, I. (1993). Systems are not only software. *Software Engineering Journal, 8,* 31-39.

Karakostas, V. (1990). The use of application domain knowledge for effective software maintenance. In *Proceedings of the IEEE Conference on Software Maintenance CSM '90* (pp. 170-176). Washington, DC: IEEE Computer Society Press.

Karakostas, V. (1992). Intelligent search and acquisition of business knowledge from programs. *Journal of Software Maintenance: Research and Practice, 4,* 1-17.

Keller, R. (1983). *The Practice of Structured Analysis.* Englewood Cliffs, NJ: Prentice Hall.

Keller, B. J. & Nance, R. E. (1993). Abstraction refinement: A model of software evolution. *Software Maintenance: Research and Practice, 5,* 123-145.

Kerr, J. & McGovern, T. (1991, October). The three R's of IS: Demystifying the reverse engineering revolution. *Database Programming and Design, 4,* 19-21.

Keyes, J. (1992, June). Code trapped between legacy, object worlds. *Software Magazine, 12,* 39-41, 44-45.

Khan, J. I. (1994). Design extraction by adiabatic multi-perspective abstraction. In H. A. Müller & M. Georges (Eds.), *Proceedings of the 1994 IEEE International Conference on Software Maintenance* (pp. 191-200). Los Alamitos, CA: IEEE Computer Society Press.

Kozaczynski, W. (1990). Basic assembler language software re-engineering workbench (BAL/SRW). *Proceedings of the IEEE Conference on Software Maintenance CSM '90,* 215).

Kozaczynski, W., Letovsky, S., & Ning, J. (1991). A knowledge-based approach to software system understanding. *Proceedings of the 6th Knowledge-Based Software Engineering Conference*, 162-170.

Kozaczynski, W., Ning, J., & Engberts, A. (1992). Program concept recognition and transformation. *IEEE Transactions on Software Engineering, 18,* 1065-1075.

Kozaczynski, W. & Wilde, N. (1992). On the re-engineering of transaction systems. *Journal of Software Maintenance: Research and Practice, 4,* 143-162.

Laffick, B. W. (1993). A programming plans paradigm for a novice programmer's support environment. *Dissertation Abstracts International, 54,* 3710B-3711B (University Microfilms No. DA9332821).

Lano, K., Breuer, P. T., & Haughton, H. (1993). Reverse-engineering COBOL via formal methods. *Software Maintenance: Research and Practice, 5,* 13-25.

Lanubile, F., & Visaggio, G. (1993). Function recovery based on program slicing. *Proceedings of the IEEE Conference on Software Maintenance CSM '93*, 396-404.

Layzell, P. J. & MaCaulay, L. A. (1994). An investigation into software maintenance--perception and practices. *Journal of Software Maintenance: Research and Practice, 6,* 105-120.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE, 68,* 1060-1076.

Lenihan, W. F. (1993, October). Refurbishing legacy systems. *Data Management Review*, *3,* 21-23.

Lerner, M. (1991, Summer). A standard approach to the process of re-engineering long-lived systems. *CASE Trends, 3,* 18-23.

Lientz, B. P. & Swanson, E. B. (1980). *Software maintenance management: A study of the maintenance of computer application software in 487 data processing organizations.* New York: Addison Wesley.

Lientz, B. P., & Swanson, E. B. (1981). Problems in application software maintenance. *Communications of the ACM, 24,* 763-769.

Lim, P. A. (1986). *CICS/VS Command level with ANS COBOL examples*. New York: Van Nostrand Reinhold.

Lukey, F. J. (1980). Understanding and debugging programs. *International Journal of Man-Machine Studies, 12,* 189-202.

Maggiolo-Schettini, A., Naoli, M. A., & Tortora, G. (1988). Web structures: A tool for representing and manipulating programs. *IEEE Transactions on Software Engineering, 14,* 1621-1639.

Markosian, L., Newcomb, P., Brand, R., Burson, S., & Kitzmiller, T. (1994). Using an enabling technology to reengineer legacy systems. *Communications of the ACM, 37,* 58-70.

Martin, J., & McClure, C. (1983). *Software Maintenance: The Problem and Its Solution.* Englewood Cliffs, NJ: Prentice-Hall.

Mattison R. (1993, October). Mattison Avenue [Column]. *Data Management Review, 3,* 32, 34.

Mayrhauser, A. von, & Vans, A. M. (1994). Comprehension processes during large scale maintenance. *Proceedings of the 16th IEEE International Conference on Software Engineering*, 39-48.

Mays, R. G. (1994). Forging a silver bullet from the essence of software. *IBM Systems Journal, 33,* 20-45.

McCabe, T. J. & Williamson, E. S. (1992, April 15). Tips on reengineering redundant software. *Datamation, 38,* 71-74.

McLoughlin, F., Estdale, J., & Tobin, M (1993). Diagramming techniques. In H. J. van Zuylen (Ed.), *The REDO compendium: Reverse engineering for software maintenance* (pp. 139-149). Chichester, England: John Wiley & Sons.

Meekel, J. & Viala, M. (1988). Logiscope: A tool for maintenance. *Proceedings of the IEEE Conference on Software Maintenance CSM '88* (pp. 328-334). Washington, DC: IEEE Computer Society Press.

Miller, J. C. & Straus, B. M., III. (1987). Implications of automatic restructuring of COBOL. *ACM SIGPLAN Notices, 22,* 76-82.

Miller, R. L. (1995a). Information Engineering: A balanced approach to information systems requirements analysis and design. *IEEE National Aerospace and Electronic Systems Conference*, 672-679.

Miller, R. L. (1995b, September). Information systems requirements analysis and design: A balanced approach. *IEEE Aerospace and Electronic Systems Magazine*, *10,* 27-32.

Miller, R. L., & Morley, J. (1996). Geriatric systems: The need for reverse engineering. *IEEE National Aerospace and Electronic Systems Conference*, 497-504.

MIL-STD- 498. (1994, December). *Software development and documentation.* Washington, DC: Department of Defense.

Müller, H. A., Tilley, S. R., Orgun, M. A., Corrie, B. D., & Madhavji, N. H. (1992) A reverse engineering environment based on spatial and visual software interconnection models. *ACM SIGSOFT Software Engineering Notes, 17,* 88-98.

Munro, M. (1992). Software maintenance, reuse and reverse engineering. In P. A. V. Hall (Ed.), *Software and reverse engineering in practice* (pp. 573-584). London: Chapman & Hall.

Napier, B. (1991, Summer). Software review: Source/RE, CGI Systems, Inc. *CASE Trends, 3,* 40-42.

Ning, J. Q., Engberts, A., & Kozaczynski, W. A. (1994). Automated support for legacy code understanding. *Communications of the ACM, 37*(5), 50-57.

Ochs, T. (1993). Cleaning out the leftovers: Software reengineering. *Software Development, 1*(6), 59-66.

Ogush, M. (1992). A software reuse lexicon. *Crosstalk: The Defense Journal of Software Engineering, 34,* 13-20.

O'Hare, A. B. & Troan, E. W. (1994). RE-analyzer: From source code to structured analysis. *IBM Systems Journal, 33,* 110-130.

Ornburn, S. B. & Rugaber, S. (1992). Reverse engineering: Resolving conflicts between expected and actual software designs. *Proceedings of the IEEE Conference on Software Maintenance CSM '92*, 32-40.

Orr, K. (1981). *Structured requirements definition.* Topeka, KS: Ken Orr & Associates.

Ostrolenk, G., Tobin, M., Altes, A., & Younger, E. (1993). The system description database and its infrastructure. In H. J. van Zuylen (Ed.), *The REDO compendium: Reverse engineering for software maintenance* (pp. 275-310). Chichester, England: John Wiley & Sons.

Ourston, D. (1989). Program recognition. *IEEE Expert, 4,* 36-49.

Partee, S. (1993, July). Data administration in the '90s. *Data Management Review, 3,* 6-8.

Partsch, H., & Steinbruggen, R. (1983). Program transformation systems. *ACM Computing Surveys, 15,* 199-236.

Peercy, D. A. (1981). A software maintainability evaluation methodology. *IEEE Transactions on Software Engineering, 7,* 343-352.

Pfrenzinger, S. J. (1992, July). Reengineering: How high and why? *Database Programming and Design, 5,* 28-35.

Pratt, T. W. (1984). *Programming languages: Design and implementation.* Englewood Cliffs, NJ: Prentice-Hall.

Price, M., Shenton, M., Davies, A., Khabaza, I., van Zuylen, H. J., & van den Bosch, P. (1993). Domain-specific issues in reverse engineering. In H. J. van Zuylen (Ed.), *The REDO compendium: Reverse engineering for software maintenance* (pp. 51-78). Chichester, England: John Wiley & Sons.

Quilici, A. (1994). A memory-based approach to recognizing programming plans. *Communications of the ACM, 37,* 84-93.

Rabin, S. (1992). Reengineering opportunities. *Computer Language, 9,* 51-53.

Raphael, B. (1966). The structure of programming languages. *Communications of the ACM, 9,* 67-71.

Rekoff, M. G. (1985). On reverse engineering. *IEEE Transactions on Systems, Man, and Cybernetics, 15,* 244-252.

Reynolds, R. G., Maletic, J. I., & Porvin, S. E. (1990). PM: A system to support the automatic acquisition of programming knowledge. *IEEE Transactions on Knowledge and Data Engineering, 2*, 273-282.

Rich, C., & Waters, R. C. (1988). The Programmer's Apprentice: A research overview. *IEEE Computer, 21,* 10-25.

Rich, C., & Wills, L. M. (1990). Recognizing a program's design: A graph-parsing approach. *IEEE Software, 7*, 82-89.

Ricketts, J. A., DelMonaco, J. C., & Weeks, M. W. (1989). Data reengineering for application systems. *Proceedings of the IEEE Conference on Software Maintenance,* 174-179.

Robson, D. J., Bennett, K. H., Cornelius, B. J., & Munro, M. (1991). Approaches to program comprehension. *Journal of Systems and Software, 14,* 79-84.

Rugaber, S., Ornburn, S. B., & LeBlanc, R. J., Jr. (1990). Recognizing design decisions in programs. *IEEE Software*, *7*, 47-54.

Sage, A. P. (1977). *Methodology for large scale systems*. New York: McGraw-Hill.

Sage, A. P. (1993). Object oriented methodologies in decision and information technologies. *Information and Decision Technologies, 19*, 31-53.

Sakthivels, S. (1994). A decision model to chose between software maintenance and software redevelopment. *Journal of Software Maintenance: Research and Practice*, *6*, 21-143.

Sammet, J. (1972). Programming languages: History and future. *Communications of the ACM, 15*, 601-610.

Sammet, J. (1981). The early history of COBOL. In R. Wexelblat (Ed.), *The history of programming languages* (pp. 199-243). New York: Academic Press.

Scherlin, W. L. (1992). A visual software process language. *Communications of the ACM, 35*, 37-43.

Shneiderman, B. (1980). *Software psychology*. Cambridge, MA: Winthrop.

Sneed, H. M. (1984). Software renewal--a case study. *IEEE Software, 1*, 56-63.

Sneed, H. M. (1991). Economics of software re-engineering, *Journal of Software Maintenance: Research and Practice, 3*, 163-182.

Sneed, H. M. (1992b). Reverse engineering versus reengineering. *Proceedings of the IEEE Conference on Software Maintenance CSM '92*, 85-86.

Sneed, H. M. & Jandrasics, G. (1987). Inverse transformation of software from code to specification. *Proceedings of the IEEE Conference on Software Maintenance CSM '88*, 102-109.

Standish, T. A. (1984). An essay on software reuse. *IEEE Transactions on Software Engineering, 10*, 494-497.

Stern, N., & Stern, R. A. (1979). *Structured COBOL Programming* 3d edition. New York: John Wiley & Sons.

Stevenson, H. P. (Ed.) (1975). *Proceedings of a symposium on structured programming in COBOL--future and present*. New York: Association for Computing Machinery.

Sullennauer, C., Olsem, M., & Murdock, D. (1992). *Re-engineering tool report.* Hill Air Force Base, UT: Software Technology Support Center.

Tamai, T. & Torimitsu, Y. (1992). Software lifetime and its evolution process over generations. In *Proceedings of the IEEE Conference on Software Maintenance CSM '92* (pp. 63-69). Los Alamitos, CA: IEEE Computer Society Press.

Teasley, B. E. (1994). The effects of naming style and expertise on program comprehension. *International Journal of Human-Computer Studies, 40,* 757-770.

Tian, J., & Zelkowitz, M. V. (1992). A formal program complexity model and its application. *Journal of Systems Software, 17,* 253-266.

Tilley, S. R., Müller, H. A., Whitney, M. J., & Wong, K. (1993). Domain retargetable reverse engineering. In *Proceedings of the IEEE Conference on Software Maintenance CSM '93* (pp. 142-151). Los Alamitos, CA: IEEE Computer Society Press.

Triance, J. M. (1978). Discussion and correspondence: A study of COBOL portability. *The Computer Journal, 21,* 278-281.

Turner, M., Neuse, D., & Goldgar, R. (1993, October). Legacy systems: Optimizing the move to client/server applications. *Data Management Review, 3,* 15-16, 18.

Ulrich, W. M. (1990a, October). The evolutionary growth of software reengineering and the decade ahead. *American Programmer, 3,* 14-20.

Ulrich, W. M. (1990b, December). From ugly legacies to artistic beauties. *Software Magazine, 10,* 33-36, 39, 42-45.

Ulrich, W. M. (1991, September/October). Business re-engineering and software re-engineering: The relationship and impact. *CASE Trends, 3,* 35-38.

Wagner, H. (1980). Visualization of structures and traces of software systems (Tool AURUM). In R. Ebert, J. Lügger, & L. Goecke (Eds.), *Practice in software adaptation and maintenance* (pp. 167-180). Amsterdam: North-Holland.

Walker, H. M. (1994). *The limits of computing.* London, England: Jones and Bartlett.

Ward, M. (1993). Abstracting a specification from code. *Software Maintenance: Research and Practice, 5,* 101-122.

Warden, R. (1992a). Re-engineering--a practical methodology with commercial applications. In P. A. V. Hall (ed.), *Software reuse and reverse engineering* (283-305). London: Chapman and Hall.

Warren, S. (1982). MAP: A tool for understanding software. *Proceedings of the 6th International Conference on Software Engineering,* 28-37.

Welch, P., & Grumman, J. (1993). The business case for reverse engineering. . In H. J. van Zuylen (Ed.), *The REDO compendium: Reverse engineering for software maintenance* (pp. 35-41). Chichester, England: John Wiley & Sons.

Weinberg, G. M. (1971). *The psychology of computer programming.* New York: Van Nostrand-Reinhold.

Weinberg, G. M. (1982). *Rethinking Systems Analysis and Design.* Boston: Little, Brown.

Weinman, E. (1991, Summer). Fighting the maintenance blues [Editorial]. *CASE Trends, 3,* 6.

Weiser, M. (1982). Programmers use slices when debugging. *Communications of the ACM, 25,* 446-452.

Weiser, M., & Shneiderman, B. (1987). Human factors of software design and development. In G. Salvendey (Ed.), *Handbook of human factors* (pp. 1398-1415). New York: John Wiley & Sons.

Wilde, N., Gomez, J. A., Gust, T., & Strasburg D. (1992). Locating user functionality in old code. *Proceedings of the IEEE Conference on Software Maintenance CSM '92,* 200-205.

Winograd, T. (1979). Beyond programming languages. *Communications of the ACM, 22,* 391-401.

Yang, H. (1991). The supporting environment for a reverse engineering system--the maintainer's assistant. *Proceedings of the IEEE Conference on Software Maintenance CSM '91,* 13-22.

Younger, E. (1993). Documentation. In H. J. van Zuylen (Ed.), *The REDO compendium: Reverse engineering for software maintenance* (pp. 111-121). Chichester, England: John Wiley & Sons.

Yourdon, E. (1989a). *Modern structured analysis.* Englewood Cliffs, NJ: Prentice-Hall.

Yourdon, E. (1989b, April). RE-3, Part 1: Re-engineering, restructuring, reverse engineering. *American Programmer, 2,* 3-10.

Yu, D. (1991). A view on three R's (3Rs): Reuse, re-engineering, and reverse engineering. *ACM SIGSOFT Software Engineering Notes, 16*, 69.

Zuylen, H. J. van. (1993). Understanding in reverse engineering. In H. J. van Zuylen (Ed.), *The REDO compendium: Reverse engineering for software maintenance* (pp. 81-92). Chichester, England: John Wiley & Sons.

Zuylen, H. J. van, & Estdale, J. (1993). Views, representations and development methods. In H. J. van Zuylen (Ed.), *The REDO compendium: Reverse engineering for software maintenance* (pp. 93-109). Chichester, England: John Wiley & Sons.

Zvegintzov, N. (1982, April). The eureka countdown. *Datamation, 28*, 172-178.