NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN INVESTIGATION OF THE
METHODOLOGY FOR SOFTWARE TRANSLATION
FROM PASCAL TO C OF AN
UNDOCUMENTED MICROCOMPUTER PROGRAM

by

Charles W. Bell

March, 1990

Thesis Advisor: LCDR Rachel Griffin

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified

1b. RESTRICTIVE MARKINGS:

2a. SECURITY CLASSIFICATION AUTHORITY:

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE:

3. DISTRIBUTION / AVAILABILITY OF REPORT: Approved for public release; Distribution is unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S):

5. MONITORING ORGANIZATION REPORT NUMBER(S):

6a. NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School

6b. OFFICE SYMBOL (If applicable): 037

7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School

6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000

7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000

8a. NAME OF FUNDING / SPONSORING ORGANIZATION: Defense Systems Management College

8b. OFFICE SYMBOL (If applicable):

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER:

8c. ADDRESS (City, State, and ZIP Code):
Director, DSS Directorate (DRI-S)
Defense Systems Management College
Fort Belvior, VA 22060-5426

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): AN INVESTIGATION OF THE METHODOLOGY FOR SOFTWARE TRANSLATION FROM PASCAL TO C OF AN UNDOCUMENTED MICROCOMPUTER PROGRAM

12. PERSONAL AUTHOR(S): Bell, Charles W.

13a. TYPE OF REPORT: Master's Thesis

13b. TIME COVERED: FROM ________ TO ________

14. DATE OF REPORT (Year, Month, Day): March 1990

15. PAGE COUNT: 120

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Software Maintenance, Software Translation, Inverse Transformation Methodology, Undocumented Microcomputer Program, Software Reusability,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this thesis is to investigate software reusability applications and the practical utilization of those applications in the performance of software maintenance. The translation of a functioning program from one high level language to another was selected as the type of software reusability effort to be explored. Five translation methodologies were investigated and the inverse transformation methodology was chosen to exercise the practical application of software reusability for a specific case study. A design strategy and translation approach was developed based on the inverse transformation methodology. The translation approach was followed in performing the translation of the case study. The results of the application of the methodology to the case study is described and the methodology is evaluated on its usefulness as a tool for software reuse.

(KR)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION: Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL: LCDR Rachel Griffin

22b. TELEPHONE (Include Area Code): (408) 646-2073

22c. OFFICE SYMBOL: CS/gr

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

18. Subject Terms (continued)

Transformation Based Maintenance Model, Attribute Grammar Technology, Automated Source Code Translators

Approved for public release; distribution is unlimited.

An Investigation of the Methodology for Software Translation From PASCAL to C of an Undocumented Microcomputer Program

by

Charles W. Bell

Lieutenant Commander, United States Navy

B.S., United States Naval Academy, 1978

Submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL

March 1990

Author: Charles W. Bell

Charles W. Bell

Approved by: Rachel Griffin

Rachel Griffin, Thesis Advisor

Daniel R. Dolk

Daniel R. Dolk, Second Reader

Acting Chair

For David Whipple, Chairman

Department of Administrative Sciences

ABSTRACT

The purpose of this thesis is to investigate software reusability applications and the practical utilization of those applications in the performance of software maintenance. The translation of a functioning program from one high level language to another was selected as the type of software reusability effort to be explored. Five translation methodologies were investigated and the inverse transformation methodology was chosen to exercise the practical application of software reusability for a specific case study. A design strategy and translation approach was developed based on the inverse transformation methodology. The translation approach was followed in performing the translation of the case study. The results of the application of the methodology to the case study is described and the methodology is evaluated on its usefulness as a tool for software reuse.

TABLE OF CONTENTS

I. INTRODUCTION

A. DISCUSSION

New software design and development costs are spiraling upward to the point where they will exceed the cost of the hardware. When the costs of maintaining the software are also considered, life cycle software costs constitute the largest portion of automated system costs [Ref. 1]. The demand for more and increasingly complex software already outstrips the capability of programmers to produce it, and the gap is expected to widen in the foreseeable future. This bleak picture is the major motivating factor behind finding ways to reduce costs and make the most efficient use of limited programmer resources. Software reusability addresses cost and resource limitations. The reuse of already developed software has become an important area of research for software developers and is receiving more attention by software application purchasers.

B. METHODOLOGY

This thesis approaches the software translation case study in three steps:

- Understanding the program

- Determining the translation methodology
- Establishing the design specification

This case study adheres to the strictest definition of software translation. The case study does not include the correction of program flaws or upgrades to the program. The need to correct program flaws or make program upgrades is often an overriding factor in decisions to initiate maintenance, but for this case study it was assumed that the target program is both functional and useful in its present form. The software translation was performed due to a change in the operating system requirements. Further upgrade or modification of the program that may be desired is defined as a separate maintenance effort and is not addressed by this case study.

The first step in software translation is understanding the program. This step assumes that the translator has no prior knowledge of or experience with the original application, a common circumstance in addressing software maintenance. The translator must acquire an overall understanding of what the application does, determine the application's level of modularity, become familiar with the variables used, and define all of the input and output. Equally important is an understanding, from the user's perspective, of how the program is used. Gaining this understanding entails more than just a study of the source code, and includes reviewing or recreating the early software development life cycle (SDLC) phases of the application. The SDLC will be discussed in more detail in future chapters.

The second step is determining the translation methodology. This step examines and analyzes several methodologies and their applicability to the case study. Determination of the translation methodology must be completed before establishing the design specifications because most translation strategies require that the design specifications be tailored to the specific needs of that strategy. The following methodologies were considered:

- Manual Re-implementation
- Attribute Grammar Technology
- Inverse Transformation
- Transformation Based Maintenance Model
- Automated Source Code Translator

Based on an evaluation of these methodologies, a specific strategy was defined which establishes the basis for the software translation.

The third step, establishing the design specifications, explicitly defines and documents the application in the form required by the selected translation methodology. The translated code is built directly from the design specifications.

The result of this effort, the translated program, is supported by the design documentation, a programmer's guide and a user's guide. The programmer's guide, including the design documentation, is a stand alon locument that describes the translated code to support maintenance and future development efforts. The user's guide is constructed for immediate use by program users.

II. BACKGROUND

A. RELATIONSHIPS AND DEFINITIONS

1. Software Reusability

Software reusability is a simple concept in theory. Since the word "reuse" means to use something more than once, "software reuse" should imply using software more than once. The immediate question then becomes "How is software defined in the context of reusability?". There are a number of definitions which have been proposed by researchers and programmers for software reusability. There is also the absence of a definitive description of what should be considered for reuse and little consensus by researchers on terminology or methodology.

A narrow definition of software reusability is the reuse of code. Code can be reused in a number of ways: using previously developed library routines in a new program; porting functions without major changes from one program or system to another; and translating a program or a portion of a program from one environment to another [Ref. 2]. Expand this limited definition to include application generators.

An application generator is software that generates new code. Therefore, using an application generator more than once is software reuse.

Restricting software reusability to code is still too limiting. The software development life cycle should not be excluded from consideration. Every phase of development from the requirements analysis to implementation and maintenance should be examined. Methodologies have been developed to reuse phases from one development effort in another effort. This, too, is software reusability. Where is the line drawn? What is reusable and what is not? If it is reusable, then how and when should it be reused? There are no definitive answers.

Given this broader scope, applications of software reusability have been categorized in a number of ways. Common categories are:

- Commercial software packages
- Code fragments
- Application generators
- Requirements analysis
- Design specifications

The above list was compiled from articles by Horowitz [Ref. 2] and Jones [Ref. 1] and is not comprehensive. Any computer language based software development tool used more than once meets this broad definition of software reusability.

Commercial software packages, also called off-the-shelf software, are not usually associated with the idea of software reusability. However, the use of off-the shelf operating systems, compilers, and general applications such as spreadsheets, word processors, and data base managers are intended to save development time, dollars, and programmer effort. The software is developed once, is centrally maintained, and is immediately compatible to varying degrees on a variety of systems. Any off-the-shelf software used in the development of more than one system can be considered reusable software. [Ref. 1]

Code fragments include library subroutines, small and large subsystems, and entire programs. The use of subroutines and subsystems range from organization specific code that is reusable only on a particular system to generic routines that are independent of its environment. High level languages such as ADA and C were designed to encourage the development and use of generic routines. These routines can be included in any program written in that language. Many of these routines are part of the standard library of functions commonly provided with that language's compiler. Some organizations also maintain a database of local routines specific to that organization. These routines are unique to the organization's particular software and hardware architecture. Entire programs are reused when they are translated to a new environment. Environmental changes generally entail translating a program to a new language or recompiling it for execution on different hardware. The key is to preserve code in a form that can be reused. [Ref. 3]

An application generator is a software product used to generate other software programs. Originally, application generators were too complicated for non-programmers and had very limited usefulness. The code produced by early application generators was extremely inefficient and required additional manual programming effort before the code could be used. Application generators are becoming increasingly sophisticated, using non-procedural languages to provide a non-technical interface with the user. They are also becoming more versatile in being able to create programs for a variety of requirements. The user enters information into the system as prompted by the generator and then the application generator produces an executable program. The created program is bug-free, eliminating the usual debugging effort, and future modifications can be made using the application generator. Programs created with application generators are still inefficient, and there are few commercial systems capable of handling large, complex software requirements.

The most important categories of software reusability are requirements analysis and design specification, the first two phases of the software development life cycle. To appreciate the importance of the reusability of these two phases, a description of the software development life cycle is necessary.

The software development life cycle (SDLC) defines the steps to develop a software program, beginning when a need is recognized. There is no standard, universally accepted SDLC. The SDLC presented in this thesis represents one approach. The SDLC phases are:

- Requirements Analysis
- Design Specifications
- Coding and Testing
- Implementation
- Maintenance

The specific steps within each phase are listed in Figure 1 [Ref. 4].

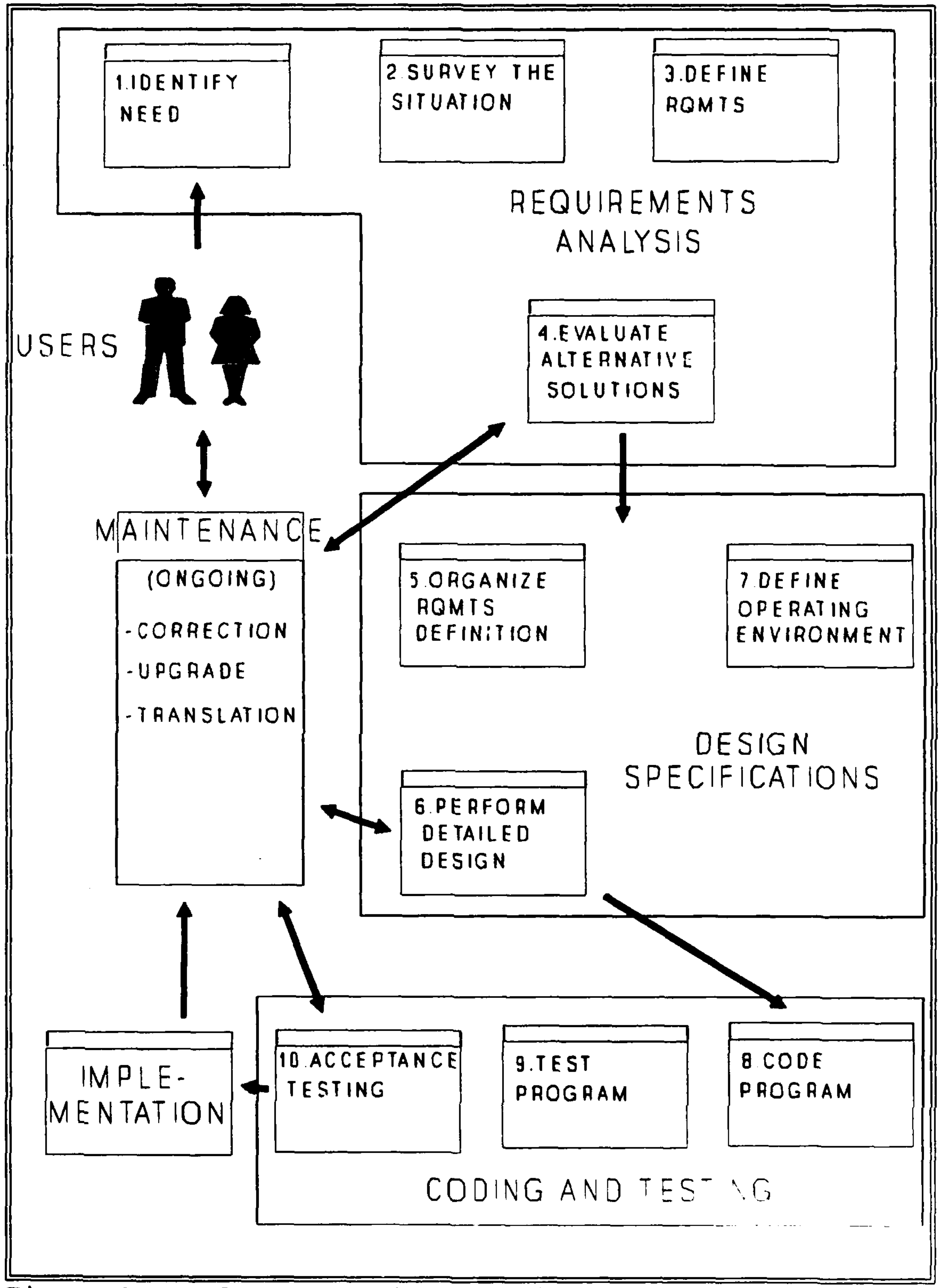

Figure 1 - Software Development Life Cycle

The requirements phase is initiated by identifying a need. This need can address a problem, an opportunity, or a directive. It can come from a user specified request, a mandate by the organization or higher level authority, or other source. Once identified, the need becomes a requirement. The requirement must be carefully defined in terms of exactly what functions are required without getting into specifics on the type of hardware or software. The requirement definition includes background on why it is needed, the advantages development of this requirement would provide, the resources that would have to be committed, and the impact of not developing the requirement. If the requirement is accepted for further development, the requirements definition becomes the baselire from which all future development phases are dependant. The information gathered during the requirements analysis is formalized in a requirements statement. Errors at this early level of development are compounded and magnified in the later phases if not corrected.

Design specifications are the blueprint the programmer works from to produce the code. Design specifications are created almost entirely from the requirements definition.

Preliminary design organizes the requirements definition in a manner suitable for computer execution. Detailed design specifications describes details about file structure, data descriptions, and program flow [Ref. 5]. Errors introduced in this phase may not be detected until acceptance testing is done. Correcting these errors can require significant recoding effort.

Coding is done directly from the design specifications. Testing is done on individual modules as they are completed and on the entire program as the modules are integrated. Program testers are concerned with ensuring successful compilation and compliance with the design specification. Acceptance testing is performed with the user and other organizational representatives to validate that the program meets user requirements. It is not uncommon for programs that fully comply with the requirements definition to still fail the acceptance test. Incomplete or poorly defined requirements and inaccurate design specifications lead to problems during acceptance testing.

Implementation involves the completion of user manuals, the training of users, and installation of the new program in the actual operating environment. Software maintenance functions consist of correcting program flaws, upgrading programs with improved capabilities, and translating programs into a new form because of changes in hardware, operating systems, technology, or language. Software maintenance once was considered independent of the development life cycle. The functions of maintenance were rarely addressed during development. Once implementation was complete, the products of development other than the completed program were not used to aid maintenance. Software maintenance is now included as part of the system development life cycle for two reasons. Foremost is the fact that more than two-thirds of the total cost of a system, from inception to scrap heap, is spent on maintenance [Ref. 1]. The inclusion of software maintenance as part of the development life cycle focused attention on these osts. Second, valuable information (documentation and lessons learned) from the other phases of development is being lost in the maintenance phase. This information has proven useful in lowering the huge cost of maintenance.

Software developers are spending increasing time on the requirements analysis and design specifications SDLC phases because of the previously noted ripple effect that errors and oversights have on later phases. Additionally, developers want to be able to maximize the quality of their high level development effort by reusing successful early development phases during the maintenance phase and with other projects. The potential for software reusability can be improved by formalizing and standardizing the requirements and design phases of the SDLC. Specific examples of this process are discussed in Chapter IV and include inverse transformations, the transformation based maintenance model, and attribute grammar technology.

There are a number of problems related to software reusability. A software developer who desires to reuse software must be able to locate reusable products, appraise their usefulness, discover any modifications that are necessary to adapt the reusable product, and evaluate the impact of using a reusable product on later phases of development. None of these basic steps can be readily accomplished at present. Although numerous libraries of reusable code are available, there is no standardized method of identifying what the reusable product does or what restrictions it may have. Trying to figure out what reusable code might be useful and what it does is similar to perusing a computer bulletin board for microcomputer programs. There are thousands of programs available, but there is no way to know which ones are best or even useful. The requirements analysis and design specification phases of development are lacking even rudimentary libraries of reusable products, although there is an abundance of successful software development efforts that would be invaluable if they could be effectively reused.

2. Software Maintenance

Software maintenance has been previously described as a critical phase of the software development life cycle that accounts for more than two-thirds of the total life cycle cost. While software reusability applications are useful development tools in the earlier phases of the SDLC, it is during the maintenance phase that the most significant benefits can be gained. Since software maintenance presupposes an existing, implemented program, by definition all software maintenance reuses software to some degree.

There are three basic kinds of software maintenance: correcting program flaws; upgrading programs with improved capabilities; and translating programs into a new form. Correcting post-implementation flaws remove problems that detract from the program's basic functionality as defined in the requirements analysis. The entire implemented program, minus the flaws, is reused. There are no changes in the requirements or the operating environment. Upgrading a program is adding functionality not addressed by the original requirements analysis. There is no change to the operating environment, but there is a change in the user requirements. Adding functionality can be the simple inclusion of a new routine with little impact on the rest of the program, but it is more likely that an upgrade will require related changes to other areas of the program. A wider range of software reusability applications pertain in this case, including the use of application generators to regenerate the program with the new functionality and methodologies which reuse the requirements or design phases. Software translation into a new form is discussed in the next section.

3. Software Translation

Software translation is necessitated by changes in the operating environment. Operating environment changes include changing the language in which the program is written, the operating system software, or the hardware. Unless the affected program is going to be redeveloped from scratch in the new environment, software reusability applications are an essential tool of this type of maintenance. One type of translation is the translation of a program from one high level language to another. The reasons for translating a program to another language vary. They include improving efficiency, improving readability for maintenance, conforming to new standards, and taking advantage of desired features in a different language. It is not a prerequisite that the targeted application be outdated, bug ridden, or otherwise flawed for program translation to be viable, although these reasons are often the impetus for consideration.

4. Summary and Purpose

Figure 2 summarizes the relationship between software maintenance, reusability, and translation. Software maintenance is the final phase of the software development life cycle. Software maintenance receives particular attention because of the disparate percentage of life cycle costs associated with performing this phase. Software reusability applications hold promise to reduce software maintenance costs. In particular, software reusability applications can be used to support software translations.

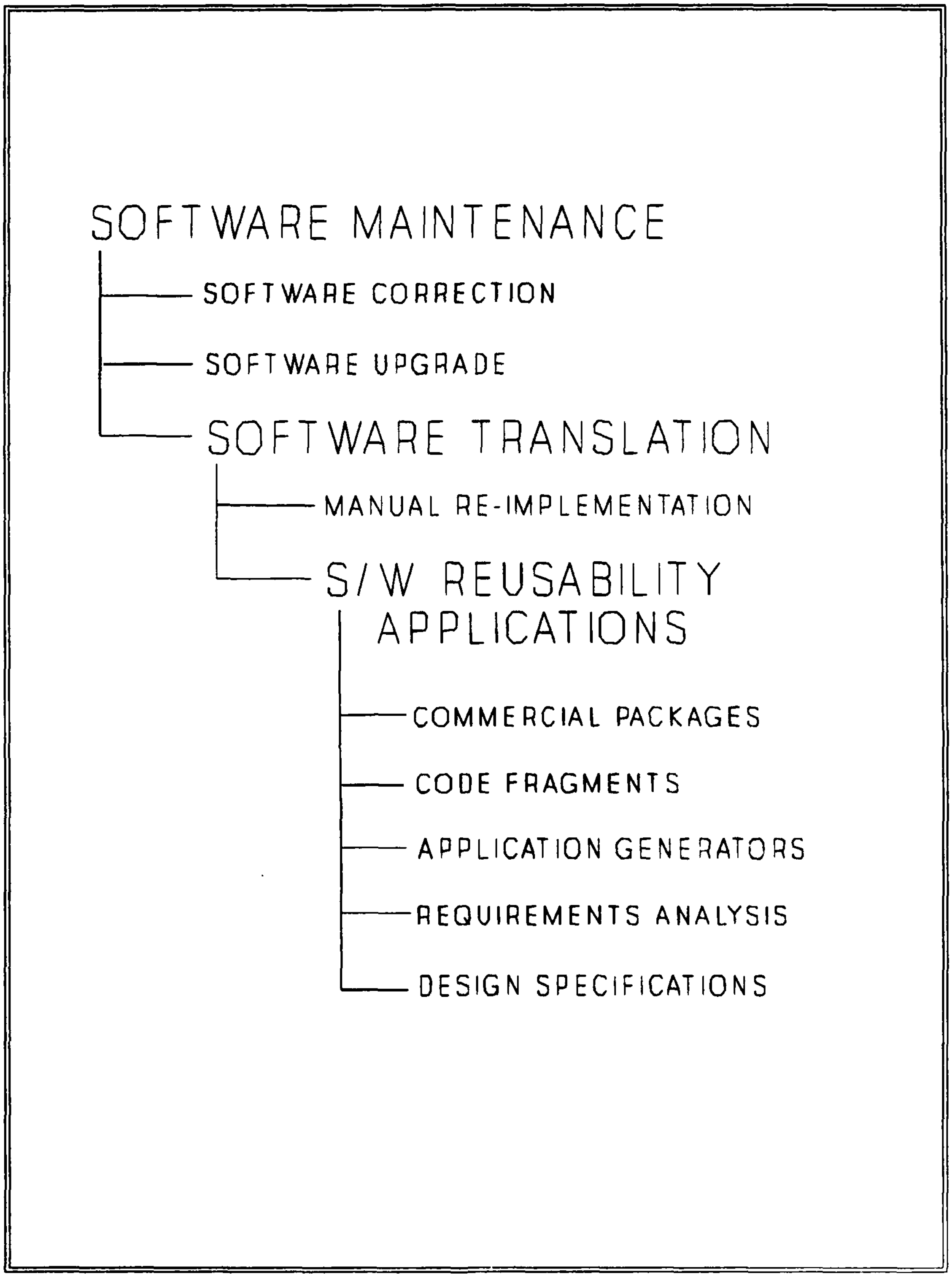

Figure 2 - Relationships

The purpose of this thesis is to investigate software reusability applications and the practical utilization of those applications in the performance of software maintenance. Of particular interest is the use of the design specifications phase of the SDLC as the primary vehicle for reuse. The software translation of a previously developed microcomputer program from one high level language to another was chosen for the case study.

Of critical importance in the reuse of early SDLC phases is the thoroughness of requirements and design documentation support. Without thorough documentation to support the application's further development, software maintenance is too hard. Unsupported software stagnates into uselessness, necessitating a costly new development effort. Software maintainers desiring to use software reusability applications have been frustrated by a lack of documentation support. They have frequently found that the necessary documentation does not exist, the personnel who developed the program are no longer available for interview or familiar with the application, and that little information remains pertaining to the development life cycle but the source code. In light of these observations, proponents of software reusability as a maintenance tool must also address the issue of how to understand the software desired for reuse and document that understanding in a manner that supports its reusability.

B. DESCRIPTION OF THE APPLICATION

1. Application Sponsor and Customers

The Defense Systems Management College (DSMC) trains military acquisition managers. Sources of students include Department of Defense program management office personnel and numerous other government and civilian organizations in the defense acquisition community. DSMC focuses its training efforts on education and research relating to program management, systems acquisition, and defense acquisition management. Program managers of major defense systems are required by Congress to attend DSMC. Customers of the parent application come from the organizations of alumni of the College.

2. Description of the Parent Application

The Program Manager's Support System (PMSS) was developed by contracted software developers under the guidance of DSMC to assist defense system program managers in acquisition program management. The primary goal of the PMSS is to improve the decision-making process for its users. The application was developed using both a top-down and bottom -

up approach. Top-down development includes development of the PMSS integrated environment and addressing issues of data compatibility and module linkage. Bottom-up development proceeded simultaneously with the development of a series of unconnected, independent modules. These modules were developed with no standardization of data format or source code language. Consequently, modules were coded in various software languages, including PASCAL, BASIC, and C. Data formats range from in-house designed data bases to the use of off-the-shelf data management software.

Faced with the growing incompatibility of the two development approaches, DSMC made the decision to reconcile the two approaches by standardizing the modules and completing development of an interface linking all of the modules into an integrated environment. DSMC also wanted future compatibility with the UNIX operating system, and so selected C as the common software language for the modules. The PMSS interface was written in C and is functionally capable of linking PMSS modules independently written in C without further modification of the module. A standardized data format has not been formally addressed but is presently under consideration.

The PMSS is composed of twenty-one modules which have reached at least the prototype stage, and ten more modules in development or being planned. The functions of these modules fall into one of seven categories: program overview/status, program impact advisor, functional analysis/support, information category data, independent modules, executive support, and utilities. The category of independent modules include all PMSS modules which have not been integrated. The Government Activity Tasking module is an independent module that has been chosen as the target application for translation.

3. The Government Activity Tasking (GAT) Module

The phrase "Government Activity Tasking" refers to procedures for providing funding from one government agency to another government agency for the performance of specified project tasks. The purpose of the GAT module is to provide the capability to track and manage project milestones (tasks) and funds assigned to other agencies. It is intended as an executive or senior-level manager module. [Ref. 6]

a. *Technical Description*

(1) Hardware. The GAT module was programmed to run on the Zenith-248 microcomputer. It requires a minimum of 384 kilobytes of random access memory (RAM) and one floppy disk drive. The module was designed for use with an Expanded Graphics Adapter (EGA) graphics hardware card with a color monitor; the use of the module with other graphics adapters or a monochrome monitor is not guaranteed. An Epson compatible printer is required to print reports.

(2) Software. The GAT module was written in PASCAL and compiled on the Borland Turbo PASCAL compiler, version 3.0. The program is broken into four chain files called in as overlays during program execution. The purpose of breaking the program into smaller sections was to keep the size of each program segment below 64 kilobytes, the maximum size limit of a PASCAL program written for the version 3.0 Borland compiler. All database support for the GAT module is provided by an off-the-shelf software product called BTRIEVE by SoftCraft Inc. BTRIEVE is executed as a RAM-resident utility program which is called by the GAT module whenever access to the database is required. BTRIEVE is automatically invoked when the GAT module is executed and is transparent to the user. In

addition to the database managed by BTRIEVE, there are program generated files which contain the information on report formats created by the report generator.

(3) Interfaces and Communications. Although the GAT module is one of many modules that make up PMSS, there is no data sharing or other communication with PMSS or any other automated system. All data used by the program is manually entered by the user. There is no requirement for a standardized database design or standard report formats. There is no requirement to provide, at the module level, any interface or link with the integrated PMSS environment.

The single interface concern is between the GAT module source code and the supporting BTRIEVE database manager. This interface is clearly defined in the BTRIEVE manual and can support the translation of the source code to C.

b. Users

The GAT module is not an operational module and is not provided as part of the integrated PMSS package. The GAT module is provided on request to defense acquisition activities desiring to beta test the module. The number of current users is unknown.

c. *Functionality*

The GAT module maintains a database of tasking information which is keyed by a task number provided by the user. When a new task number is added, all information about that task is entered on the keyboard by the user. Task information may be edited and tasks deleted whenever required. Program commands are executed primarily by the use of function keys. Most function key commands are listed in a menu which appears across the bottom of each screen. Data about each task are displayed on a three screen worksheet. The user can print a summary report of task information, single screens of the task worksheet, or the entire task worksheet using function commands. Additionally, a report generator is provided for developing and printing reports. The report generator allows the user to select which data elements of the task will be included in the report. The created report heading can be saved as a report format and the information requested can be printed for all tasks in the database.

III. UNDERSTANDING THE SOFTWARE

A. INFORMATION SOURCES

Software that is not understood cannot be maintained. Software maintainers commonly have little background or experience with the majority of programs they are tasked with maintaining. It is imperative that software maintainers acquire detailed information about the target program. Every source of information available must be examined in detail. SDLC documentation is a very important source of information, but the maintainer must exercise extreme care in reviewing this material. The maintainer must determine how closely the documentation reflects the actual program and identify those portions of the documentation that are no longer accurate. Programmer manuals and user's manual, if available, should be studied with a certain degree of skepticism. It is unusual for the manuals to be updated when changes and modifications are made to the program, and it is common practice for the manuals to be created after program completion with marginal regard for their accuracy. Program source code is also a good source of information, provided the source code listing available for use correctly represents the executable program.

If possible, the source code listing should come directly from the source code used to compile the program; any other program listing may not reflect undocumented changes made to the executable program. Other documentation that can resolve confusion or help make cryptic code more understandable is user operations documentation. User operations documentation consists of all regulations, instructions, and policies of program users that pertain to the target program. This is particularly true if the requirements documentation is not accurate or non-existent. User operations documentation normally provides much of the information the original programmers used in the requirements analysis.

Documentation is not the only source of information about a program. If available for interview, the original programmers and program users can provide important information. The maintainer should not expect the original programmers to remember details about the program. Typically a significant amount of time has passed since the programmers were directly involved with the target program, and it is unlikely that the programmers can answer detailed questions about specific lines of code. However, questions about the general structure of the program and why certain decisions about that structure were made can be very revealing. Program users can add information to support the available documentation. In cases where there are no requirements documentation or supporting user operations documentation, program user interviews may be the only way to ascertain the baseline guidance for the original development of the program.

In summary, the software maintainer should consider the following sources (compiled from articles by Phillips [Ref. 7] and Fay [Ref. 8]) when collecting information in preparation for doing software maintenance on an unfamiliar program:

- SDLC documentation
- Program source code
- User regulations, policies, and instructions
- Programmer manuals and user's guide
- Source code programmer interviews
- Program user interviews

The succeeding sections address the sources of information explored in developing an understanding of the GAT module.

1. Available Documentation

The documentation available on the GAT module is very poor which, as previously discussed, is typical of most applications. The GAT module program documentation consists of a source code listing and a user's manual. The source code listing is available both as a hard copy list and on diskette as compilable source code. The availability of the source code on disk guaranteed that the source code matched the executable program. The executable program was produced by compiling the source code. A new listing was printed from this same code to ensure that all source code information accurately reflects the executable program. The user's manual is a better than average product which effectively teaches the user the operation of each program option. The user's manual is straightforward and simple to use. Problems with the manual were minor, such as inaccurate information on the use of some keyboard keys and the lack of an index or a summary of available functions.

User operations documentation is non-existent. The requirements for the original program were collected by interviewing potential users. No record of these interviews exist.

Other documentation that is not available for the GAT module is the programmer's manual and any documentation relating to the SDLC of the program. Requirements analysis documentation and design specifications were not created when the program was developed, and there is no documented record of any subsequent changes made.

2. User's and Programmers

No GAT module users were available for interview. The programmers for the contracted software development organization which developed the GAT module were queried for general information on the development of the module. Conversations with the organization revealed the lack of documented support for the module and the dependance on third party software to generate much of the code. No major insights on the development were revealed and, as expected, little detailed information could be provided.

B. PROGRAMMER AIDS

Programmer aids refer to the software tools and analysis methodology used by the maintainer to understand and translate the GAT module. Fundamental tools which comprise every programmer's basic toolbox, such as a computer, a program editor, and a compiler, are not addressed. Instead, emphasis is given to tools which specifically aid in deciphering source code and translating between high level languages.

1. Automated Tools

Two software tools were used to increase the maintainer's understanding of the source code. One software tool, Tree Diagrammer by Powerline Software, provided the translator with detailed information about each procedure and function (also referred to as "routines") defined and used in the program. The second tool, Source Print, also by Powerline Software, provided a comprehensive listing of all variables used.

Two software products were used to manage data and generate code for the GAT module by the original software developers. One software tool, Softcode by The Software Bottling Company, is an application generator which produces code for screen layout. The other software tool, BTRIEVE by Novell, is a RAM resident utility which provides data base management functions. Learning to use these tools was necessary to gain an understanding of their role in the development and execution of the program.

a. *Deciphering Source Code*

The maintainer initially attempted to define the logical control flow of the program by manually studying the source code and making lists of routines and variables. This method was lengthy, tedious, and resulted in many errors. Tree Diagrammer was then used. Tree Diagrammer thoroughly mapped all calling routines in graphical format, clearly described dependencies and flagged anomalies. The level of nesting of other routines within each routine were also defined. Tree Diagrammer proved useful because it pulled essential data out of the source code and presented that data in an effective format. This information facilitated understanding the control flow of the program. Source Print aided the development of the data dictionary by providing a listing reporting the name and location used of every variable.

The use of software tools such as Tree Diagrammer and Source Print is not a panacea for the understanding of undocumented source code. These tools automate certain processes that the maintainer must otherwise accomplish manually when preparing information needed to understand the source code. Automated tools save valuable time. It is still up to the maintainer to interpret and clarify the information generated into a clear picture of the program processes.

b. Automated Source Code Translation

Two automated translators were experimented with. Specific information on the features of the automated translators are discussed in Chapter IV.

2. Structured Systems Design

Structured systems design [Ref. 5] is a well established methodology introduced to improve the development of reliable and maintainable software systems. It is a methodology created specifically for software systems development, and is the heart of the manual re-implementation methodology described in Chapter IV. The disciplined approach of structured design also served to provide a good framework in advancing the maintainer's understanding of the GAT module. Structured design components such as structure charts, pseudocode, entity-relationship diagrams, and data dictionaries were used to represent the information gained from studying routines and variables. The construction of these components completed the source code analysis.

C. PROGRAM DETAILS

1. Structure

Program structure defines the composition of the program by modules. Modules are discrete blocks of code for which the inputs, outputs, and functionality can be described. Modules are made up of other modules in a chain that begins with the program as a whole and ends with simple modules which cannot be further divided. The division of a program into a modular structure and the relationship between modules is called partitioning and hierarchical organization [Ref. 5].

The GAT module is constructed using overlays. The purpose of overlays is to allow the creation of programs larger than the maximum that can be accommodated in computer memory. The overlay procedure is complicated to execute, but simple to explain. Program routines are collected together into subprograms. Routines within a subprogram cannot call another routine in a different subprogram, because only one subprogram can be present in memory at a time. The GAT module has a main program and four subprograms. The main program is always present, and it ensures the appropriate subprogram is available in memory when required. Ideally, subprograms are functionally self-sufficient and do not require any of the routines needed by other subprograms. In the GAT module, however, several routines are needed by all the subprograms. This need is accommodated by duplicating the desired routines in every subprogram that requires the routines. The end result is effective but inefficient. The GAT module runs successfully and stays within the memory limits imposed by the Borland PASCAL compiler, but wastefully duplicates code and increases disk size.

Learning the structure of the program was accomplished using the following steps:

Step 1: <u>Define the overall function of the program</u>. This module is the top level module of the structure chart. "Track Tasks" was defined as the overall function of the GAT module.

Step 2: <u>Describe obvious, high level functions as modules</u>. Ask the question "What does this do?" of user decision points in the program. Menu items and function key selections are the best clues to use to gain a general idea of the main functions of the program.

Step 3: Describe the next level of functionality within modules identified in Step 2. Follow the same thought process as in Step 2. Repeat this step with subsequent module levels until the function of the module can no longer be divided without reference to specific routines.

Step 4: Define the function of each program routine. These routines are the lowest level modules. Strongly structured programs have readily definable routines which perform a single function. This is called cohesion [Ref. 5]. Most programs, however, contain routines that have multiple functionality or have no clearly definable function at all. Define all functions performed by a routine and note routines that cannot be clearly defined.

Step 5: Cross reference modules defined in Step 4 with the lowest level modules described in Step 3. Look at the function of each Step 3 module and determine which modules from Step 4 are required to perform that module. Step 4 modules can be used with as many Step 3 modules as required. Multiple function modules should be included even when some functions are not applicable.

Step 6: Correct module cohesion problems. Review the cross reference created in Step 5. Break routines that perform multiple functions into separate, cohesive modules. Eliminate undefinable routines by absorbing their functions into related modules. Revise the cross reference to reflect changes.

2. Control Flow

Control flow is the order in which modules are executed, and is represented by the hierarchical arrangement of the modules. Control flow in the GAT module was traced by studying program execution and manually walking through the source code. It is not required that control flow be represented in the structure chart. For this case study, however, control flow is defined in the structure chart and was determined concurrently with understanding program structure. The following steps were performed concurrently with the same step number used to determine structure above.

Step 3: Bond related modules together by order of execution. Define the order in which modules at the same level are executed.

Step 4: Define the order in which routines are executed and dependencies between routines.

Step 6: Correct module coupling problems. Coupling is the degree of interdependence between modules [Ref. 5]. Low coupling is desirable because modules should be independent of each other. The following should be considered: modules should not branch into the inside of another module and modules should not alter the statements of other modules.

Step 7: Develop the initial structure chart. Show the modules and the connections between the modules. Do not include the data communicated between modules at this point.

3. Variables

Variables are names used to refer to stored data. The data stored may be a single element or composite data made up of more than one component. Information about a variable includes its definition, components, the elements which make up the components, and its physical format. Variable information is displayed in the data dictionary.

The steps performed to determine variable information are:

Step 1: List all variables used in the program. The utility program Source Print was used to create the list.

Step 2: Determine variables with composite data. Identify the components and elements which make up the composite data.

Step 3: Describe each variables' physical format. Physical format is a description of the values that a variable may take on, the number of characters allowed, and the type of characters allowed. This information is available in the declaration statement of the variable.

Step 4: Determine where variable values are assigned and used. Add to module descriptions a list of variables used and variables changed by each module. Variables used by a module, unless created within the module, are the module's inputs. Variables changed by a module and subsequently used by other modules are the module's output.

Step 5: Update the structure chart. Show communication between modules by tagging module connections with the variables input and output between modules.

Step 6: Correct module coupling problems. Review the variables being passed between modules. The following should be considered: pass only variables essential to the module; minimize the use of global variables which are not passed; and minimize passing composite data if little of the data is actually used. Revise the structure chart.

4. Input Sources

Input sources provide data not initialized or calculated by the program. Input sources for the GAT module are user input from the keyboard, database files, and report format files. Modules on the structure chart representing the retrieval of this data are not as detailed as other modules because they use routines external to the program.

5. Output Destinations

Output destinations receive data for storage or display. Output destinations for the GAT module are the monitor screen, printer, and disk drive. Modules on the structure chart representing this data are not as detailed as other modules because they use routines external to the program.

D. APPLICATION INCONSISTENCIES AND RECOMMENDATIONS

Application inconsistencies noted in this section are the result of a review of the program strictly from a user friendliness and consistency of design point of view. The decision to implement any or all of the following recommendations is based on the consideration of which requirement, the status quo or the recommended change, is most consistent with the design strategy adopted and takes the best advantage of the target language, C.

1. Screen Movement

The following inconsistencies in screen movement were noted:

- Arrow keys are the primary method of moving the cursor to different areas of the screen; however, on some screens the up/down arrow combination is required and on other screens the left/right arrow combination is required. The reason for this is not logically evident.

- Similarly, the use of the PageUp/PageDown keys also vary from screen to screen.

Recommendation: Develop a consistent design which allows the use of all four arrow keys on every data entry screen. Use the logical meaning of the PageUp/PageDown keys for paging between worksheet screens.

2. Function Key Use

The following inconsistencies in function key use were noted:

- The meaning of function keys (<F5> in particular) changes depending on the screen.
- Function key <F9>, used to change screen color, does not appear in the bottom line menu and only works at certain places in program.

Recommendation: Develop a design that consistently applies the same meaning to function keys. Design bottom line menus which display all enabled function keys. Add the <ESC> key to the menu for incremental backtracking to the main menu.

3. Report Generator

The following inconsistencies in the report generator were noted:

- The form generation routine assumes that 120 column print is always used, requiring the user to do manual calculations to accommodate other sizes.

- The size of individual data items can only be displayed one at a time, complicating the process of creating a report heading.

- The <return> key is used to transfer a selected data heading to the report generator, but this is not shown in the menu.

- A predefined report format filename must always be manually entered even though a list of predefined report formats can be displayed.

Recommendation: Completely redesign the report generator to correct the above inconsistencies. This redesign is a significant departure from the original program and may not be applicable to this translation effort.

4. **Other**

The following remaining general inconsistencies were noted:

- Users are arbitrarily constrained to a limited number of lines describing the task.

- Saving changes to the task worksheet can be done only when quitting the program.

Recommendation: Develop a design to allow unlimited (except by available memory) descriptions and include a save option in the bottom line menu which can be executed during add or edit operations.

IV. SOFTWARE TRANSLATION METHODOLOGIES

A. OVERVIEW

The software translation methodologies discussed in this chapter consist of four software reusability applications and one SDLC implementation. Each methodology is described in terms of purpose, functionality, complexity, and applicability to the case study.

An important aspect in determining the translation methodology which best fits the case study is the degree of commonality of the source and target language. Languages are developed with certain strengths and weaknesses. Languages support specifically defined sets of functions that may be similar to another language or may be totally unique. It is the dissimilarities between languages that complicates the translation process. If two languages supported all the same functions in exactly the same way, then translation would be a straightforward process readily managed by an automated translator. Two languages developed with completely incompatible and unique functions may not be translatable by anything short of manual re-implementation.

The programming languages PASCAL and C lie between the two extremes. There are many similarities and some fundamental differences which must be considered in establishing a translation methodology. Prior to describing the translation methodologies considered, a comparison is made between PASCAL and C.

B. COMPARISON OF C AND PASCAL

The first determination to be made in making comparisons between two programming languages is what to compare. Both PASCAL and C have numerous compilers with varying functionality. Both languages have more than one widely popular "standard" definition. The standard definitions for PASCAL are the Wirth definition and the Borland Turbo definition [Ref. 12]. Niklaus Wirth developed the original PASCAL language [Ref. 9]. The Borland definition is an extension of the Wirth definition and is the compiler of choice for many microcomputer users. The standard definitions for C are the Kernighan and Ritchie definition and the American National Standards Institute (ANSI) definition [Ref. 10]. C was developed by Dennis Ritchie [Ref. 11]. The American National Standards Institute promotes a standard definition, ANSI C. ANSI C is an extension of the Kernighan and Ritchie definition. The Kernighan and Ritchie definition is used on the UNIX operating system.

For this case study the source program was written in Borland Turbo PASCAL and the target program was written in Microsoft C. Microsoft C supports both the Kernighan and Ritchie and ANSI C standard definitions. The language comparison will be based on the extended standards of Borland Turbo PASCAL and ANSI C.

A second issue is the C standard function library. This library includes additional functions, primarily for input/output operations, that are not part of the standard definition of C. Since this library is always included with C compilers, the functionality provided by the library was included in the comparison.

The two languages are compared in the following three categories: the history and purpose of development; comparison of features; and suitability for the case study.

1. Purpose and Goal of the Languages

C was designed in 1972 by Dennis Ritchie [Ref. 11] for the following reasons:

- To provide a computing language implementable on small machines.
- To be used to implement operating systems and language processors.

* To provide programmers with an efficient interface to computer hardware.

PASCAL was designed by Niklaus Wirth in 1969 [Ref. 9] for the following reasons:

* To provide a systematic and precise expression of programming concepts, structures, and development.
* To demonstrate that flexible language facilities can be implemented efficiently.
* To provide a good vehicle to teach programming by the inclusion of extensive error checking facilities.

"The design goals of PASCAL and C were quite different. PASCAL's restrictions were intended to encourage the development of reliable programs by enforcing a disciplined structure. By strongly enforcing these restrictions, PASCAL helps the programmer detect programming errors and makes it difficult for a program, either by accident or design, to access memory areas outside its data area.

In contrast, C's permissiveness was intended to allow a wide range of applicability. The basic language has been kept small by omitting features such as input/output and string processing. Ideally, C was to be sufficiently flexible so that these facilities could be built as needed. In practice this philosophy has worked well."

[Ref. 12]

A prominent difference in the two languages is their treatment of variable types. PASCAL is a strongly typed language. C is not. Strongly typed languages mandate that a variable can belong to only one type and that type conversion is accomplished by converting a variable value from one type to another. PASCAL limits type conversions to explicitly called routines and mixed-mode expressions containing integer and real variables. C does not always require that variables be checked for type compatibility. For example, the language definition does not require that the types of actual and formal parameters be checked for compatibility. Strongly typed languages such as PASCAL improve program clarity and reliability. Loosely typed languages such as C encourage and support programmer flexibility. [Ref. 12]

2. Comparison of Features

This section addresses the main differences in the two languages [Ref. 12]. The purpose of this section is to highlight areas of concern for software translation. It is assumed the reader has a basic familiarity with programming language concepts and the features of the two languages. A complete description of the languages is not intended.

a. Data Types

PASCAL data types provide security from errors, readability, and reliability primarily attributable to consistency checking not required by C. C data types allow addressing physical memory locations, multiple precision arithmetic, no restrictions on where pointers can point, address arithmetic, and few restrictions on manipulating arrays.

b. Statements

The C and PASCAL languages use the semi-colon in a slightly different manner. In C the semi-colon is used as a statement terminator. In PASCAL the semi-colon is used as a statement separator. The PASCAL method is more error prone because there are more conditions that determine when the semi-colon should be used than in C.

Control statements are functionally very similar. One exception is the switch (C) and case (PASCAL) statements. In C, the switch statement executes multiple alternatives in the order they appear unless an explicit transfer of control is given. In PASCAL, only one alternative is executed in the case statement. Another difference is that PASCAL has no controlled transfer statements in PASCAL such as break and continue. The lack of some analogous statements between the two languages complicates translation. The PASCAL repeat loop can be simulated in C using other constructs. The C controlled transfer statements break and continue can be simulated in PASCAL. However, the simulated constructs are not as efficient as their counterparts and make the program more difficult to understand.

c. *Program Structure*

There are significant differences in the structure of the two languages. PASCAL is a block structured, hierarchical language which supports the nesting of routines within other routines. C is considerably less structured to maximize programmer flexibility. Program structure differences are summarized as follows:

- Order of Appearance. PASCAL requires a strict order of appearance of the different parts of the program. For example, the main body of a PASCAL program must be at the end of the program. This ordering helps ensure one-pass compilation of the program but reduces program readability. In C, order of appearance is much more flexible.

- Variable Visibility. C provides very flexible methods of expanding or restricting the scope of variables, encouraging the use of shared private variables to improve reliability. PASCAL requires the use of non-local variables or strict parameter passing to get information between routines.

- Passing Parameters. In PASCAL, parameters can be passed between routines by either value or reference. C parameters can be passed only by value. In C, the address of a variable must be passed to achieve the same effect as passing by reference. PASCAL requires that the number of variables passed equal the number of variables expected by the called routine. C does not check that the number of actual parameters equals the number of formal parameters expected by the called routine.

- Entry and Exit Points. PASCAL routines must be entered and exited from the beginning of the routine and its end, respectively. In C, specific control statements such as break and continue allow entry and exit from arbitrary places within a control structure.

- External Routines and Variables. C allows the use of external routines and variables, encouraging the development of libraries of routines. The version of PASCAL used in the case study does not support external routines or variables.

C. METHODOLOGIES REVIEWED

1. Inverse Transformation

The inverse transformation methodology described by Sneed [Ref. 13] is based on the strategy of reversing the normal software development cycle. Software is viewed at three levels which are an abstraction of the output of the structured analysis and design methodology. Abstraction Levels are physical, logical, and conceptual [Ref. 13] and correspond to the source code, design specification, and requirements statement of structured analysis and design.

There are two steps in the process. The first step applies reverse engineering techniques to retranslate the source code into an intermediate design schema. The result of the retranslation is design documentation based on the intermediate design schema. The second step applies standard software engineering principles to translate the intermediate design schema into a system specification.

The objective of the inverse transformation methodology is the creation of the requirements statement.

Proponents of reverse engineering claim that viewing the software at this conceptual level improves software maintenance and reusability [Ref. 13]. Inverse transformation is not the same as software restructuring. Software restructuring is used to reduce maintenance costs by converting unstructured programs into structured programs [Ref. 13]. The application of software restructuring does not require the recreation of the requirements statement or design specification. The extent of restructuring done as part of the inverse transformation process is dependent on the rigor in which the original development was conducted. Poorly designed and unstructured programs require more restructuring than well designed programs.

Defining the transformation rules required to accomplish the first step in the inverse transformation methodology is dependant on "... the structure of the programming language as input and the structure of the design schema as output...." [Ref. 13] In other words, the translator starts with the software language of the source code, defines the design schema to be used, and then determines the transformation rules.

Transformation rules are built by inverting the process of generating code from design documentation. For example, if the design schema defined by the translator requires relational tables to describe a database, then relational tables should be created from any database described in the source code. The specific process to accomplish the first step transformation is left to the translator.

In the second step, the translator takes the design documentation from the first step and creates the system specification based on the Entity/Relationship (E/R) model. E/R models are described by Whitten [Ref. 4]. Two levels of abstraction are defined by the inverse transformation methodology, micro and macro, which represent the two levels of detail required in the E/R model. These levels are further broken down into a number of more specific specification levels defined as entities, structures, and relationships. The purpose of this breakdown is to reach a level of detail comparable to that of the design schema. Once this is accomplished, the translator links design elements and specification elements together into a set of assignment criteria that guide the retranslation from the design schema to the system specification.

The end result is a system specification that is an exact, conceptual representation of the original source code.

The system specification serves as the baseline for system maintenance and module reuse.

2. Transformation Based Maintenance Model

The Transformation-based Maintenance Model (TMM) is a methodology that allows "... practitioners to recover abstractions and design decisions that were made during implementation." [Ref. 14] TMM relies on the use of a prototype tool called Draco.

The Draco paradigm is based on the idea of a domain-specific "super" language that would map onto a real software language. Draco provides the methodology to abstract language dependant design decisions into a more generic form represented by nodes on a graph. Design decisions that are dependant on prior design decisions are linked together, and alternate methods of achieving the same design decision are shown as alternate paths on the graph. This graph, called a Directed Acyclic Graph (DAG), becomes the basis for the system specification.

There are a number of prerequisites to using TMM. The most significant and restrictive prerequisite is that the system specification must be derived from the Draco paradigm. If a Draco derived system specification is not available, it must be developed before TMM can be employed. Since few programs have been developed using the Draco paradigm, the process of employing TMM must include steps to produce the needed Draco specification. If the assumption is made that the translator does not have a Draco derived system specification, the following outlines the steps to applying TMM:

Step 1: Begin the abstraction recovery.

Step 1A: Propose abstractions from the source code.

Step 1B: Choose the most suitable abstractions.

Step 1C: Construct the specification from the chosen abstractions.

Step 2: Create the Directed Acyclic Graph (DAG).

Step 3: Identify the Least Common Abstraction (LCA).

Step 3A: Identify code that contributes to the undesired design.

Step 3B: Reverse undesired design decisions.

Step 3C: Collect undesired code into a single component.

Step 3D: Re-implement the undesired component.

Step 4: Choose the new desired path on the DAG.

Abstraction recovery is comparable to the first step in the inverse transformation methodology (retranslating source code into an intermediate design schema) previously discussed. The product of abstraction recovery is the Draco specification which supports the creation of the DAG and the identification of the LCA.

In the DAG, the top node, or root node, represents the original specification, and subsequent nodes represent correct but partial design decisions of the specification. The DAG traces possible design decisions, beginning at the root and ending when the last design decision is made. The translator uses the DAG as a translation tool by searching backward up the nodes of the DAG toward the root until a node which encompasses both the original implementation and the desired implementation is found. This node is the LCA. The LCA becomes the new starting point on the DAG to trace the path to the desired implementation. As the translator traces the path to the LCA, he reverses the design decision at each node and identifies undesired portions of the original implementation. The translator collects the undesired portions together as a single component for re-implementation, and traces a new path on the DAG from the LCA to the desired implementation.

3. Attribute Grammar Technology

The use of grammars to describe high-level programming languages is an established instrument of programming language theory and shows promise as a tool for source-to-source language translation. Attribute grammar technology is an extension of grammar based methodologies. A synopsis of commonly used terminology [Ref. 15] is provided below to support the discussion.

Grammars: Grammars formally specify the syntax of the language with a set of rules describing the set of all statements that are legal and correct in the language. A grammar imparts no meaning to the constructs it describes, only what is syntactically legal.

Statement: A statement is a source code fragment. For example, the PASCAL fragment in brackets [C := A + B;] is a statement. A statement is comprised of a sequence of tokens.

Tokens: A token is a string of characters that make up a portion of a statement. Tokens are normally keywords, arithmetic operators, variable names, etc.

Parsing: Parsing is the process of analyzing a sequence of tokens and identifying the sequence with the correct language construct described by the grammar.

Productions: Productions are the rules of grammar used when parsing to describe all the statements of the language.

Parse Tree: A parse tree is a graphical representation of the grammar of the language and is used in the analysis of a program or any portion of a program (such as a statement). See Figure 3.

Sentence to be Analyzed

THE CAPTAIN COMMANDS THE SHIP.

= =

Parse Tree

```
                           sentence
                -------------|----------
               |                        |
            subject                 predicate
         ------|------           -------|---------
        |             |         |                 |
     article        noun       verb        direct-object
        |             |         |             ------|---
        |             |         |            |          |
        |             |         |         article     noun
        |             |         |            |          |
       THE         CAPTAIN   COMMANDS       THE        SHIP
```

= =

Grammar

sentence	::=	subject predicate	(root node)
subject	::=	article noun	(nonterminal)
predicate	::=	verb direct-object	(nonterminal)
direct-object	::=	article noun	(nonterminal)
article	::=	THE	(terminal)
noun	::=	CAPTAIN	(terminal)
noun	::=	SHIP	(terminal)
verb	::=	COMMANDS	(terminal)

= =

Explanation

1. The [::=] symbol means that the items to the right of the symbol are subnodes of the item on the left of the symbol.
2. The grammar is the set of rules which specify for each nonterminal the kind and number of subnodes attached and the order they appear.
3. Grammar rules (productions) and the parse tree can be used two ways. First, all possible ways to devise a sentence can be traced. There are four possible sentences:

THE CAPTAIN COMMANDS THE SHIP
THE CAPTAIN COMMANDS THE CAPTAIN
THE SHIP COMMANDS THE CAPTAIN
THE SHIP COMMANDS THE SHIP

Second, a given sentence can be checked to determine if it is one of the possible sentences. This analysis is called parsing.

Figure 3 - Example of a Parse Tree

Context-free grammar: A context-free grammar does not take context into account in its set of rules. A statement in a program is parsed based only on the sequence of tokens that describe the statement, and does not take into account any information from the parsing of previous statements.

Context-sensitive grammar: A context-sensitive grammar considers the fact that prior statements already parsed may have an effect on the validity of subsequent statements and in the way subsequent statements are parsed. Context-sensitive grammars are more complex than context-free grammars because of the need to have access to information about prior statements.

Attribute grammar: An attribute grammar is an extension of a context-free grammar and formally specifies context-sensitive rules.

Attributes: Attributes are context-sensitive rules of grammar. Attributes are directly associated with productions and are expressed in the form of conditions which must be evaluated.

Attribute values: Attribute values are determined by evaluating attributes and associated productions.

A simple example illustrates the application of attribute grammar technology. The following are two statements in a PASCAL program. Statement #1 is a variable declaration and statement #2 uses the variable declared in an assignment statement.

```
statement #1: X : char;

statement #2: X := 1;
```

Assume that only a context-free grammar is available to analyze the two statements. Statement #1 is first scanned and parsed into tokens. A sequence of four tokens is recognized: variable name [X], operator [:], keyword [char], and operator [;]. The analysis of this sequence of tokens determines that the statement, with respect to the grammar, is legal. The same is done with statement #2, without taking into account the first statement already analyzed. Statement #2 is also determined to be legal. However, compiling these two statements with a PASCAL compiler would cause statement #2 to be flagged as an error. Variable X was declared to be of type char (character), but was assigned an integer value, which is illegal in PASCAL.

How did the compiler recognize the error? This is a context-sensitivity issue. Using attribute grammar technology, attributes are inserted into the grammar which cause additional analysis of the sequence of tokens. The analysis then includes steps that recognize statement #1 as a variable declaration, checks a symbol table, and returns a value that indicates if X has been previously declared. In statement #1 X has not yet been used; variable X is added to the symbol table and statement #1 is accepted as a legal construct. When statement #2 is analyzed it is recognized as an integer assignment. Evaluation steps are performed which checks the symbol table and returns a value indicating variable X is defined, but not as an integer as required by the attribute grammar. Statement #2 is flagged as an error. The important difference is that in order for statement #2 to be evaluated properly, prior knowledge about statement #1 was necessary.

An attribute grammar will not work with a language for which it was not specifically constructed. Revising the above example, the same two statements are written in C as follows:

statement #1: char X;

statement #2: X = 1;

Although the string of characters are largely the same, the PASCAL operator [:=] has been replaced by the C equivalent operator [=] and the sequence of tokens has changed. A different context-free grammar and attribute grammar is necessary to describe the language. Using the same chain of logic described above, statement #1 would be recognized as legal and not statement #2 because of the type mismatch. However, in C the type char is only another representation of the type integer. In C integers can be assigned to variables of type char without error. In reality, both statements are legal in C. The attribute grammar must reflect this properly.

An attribute grammar is very specific to the language it describes. In order to use attribute grammars for language translations, an intermediate language is needed to bridge the differences in the languages. An attribute grammar is developed which translates the source language to this intermediate form, and another attribute grammar is developed to translate the common intermediate form to the target language.

The intermediate form is devised in one of two ways, the greater common devisor method and the least common multiple method. When using the greatest common devisor, the translator attempts to create an intermediate form that retains as much of the higher level functions of the two languages as possible. In order to represent functions that exist in one language and not the other, these high level functions are rewritten as a series of lower level functions that are common to both languages. This causes inefficiencies and loss of program structure if not used carefully. The greatest common divisor method works well with source languages that are closely related, such as C and PASCAL. It is less successful with languages in which the syntax is disparate because many low level constructs are needed to commonly represent the two languages. A second method, the least common multiple method, addresses this issue by requiring the development of attribute grammars for both the high level function and its low level constructs for every disparate function. Although the least common multiple method minimizes translation inefficiencies in dissimilar languages, there is a corresponding increase in the complexity and level of effort required to develop the attribute grammars. [Ref. 16]

There are some language constructs which cannot be represented by attribute grammars. One example of this is complex pointer arithmetic commonly used in C. Such non-representable constructs are flagged without translation. A different translation methodology for these constructions is necessary to complete the translation. [Ref. 16]

4. Manual Re-implementation

Manual re-implementation is the development of the program as if no previous program existed. The full software development life cycle is performed. The requirements statement is generated from user defined requirements and a study of the current environment. The source code of the previous program and all implementation decisions and other information arising from the development of the earlier program is ignored. Based on the new requirements analysis, the remaining steps of the SDLC are performed.

5. Automated Source Code Translation

Automated source code translators take the source code of the original program as input and output source code for the translated program. Automated translators are rated in four areas:

- Effectiveness of syntactic conversion of like functionality
- Degree to which unique functional differences are addressed
- Efficiency in converting unique functions to similar constructs
- Overall effectiveness of the translation

Automated translators vary in the degree in which language differences are addressed. A minimally successful automated translator must correctly convert all like functions between the source and target language and should flag code that the translator could not convert. For example, converting the assignment statement in PASCAL to the assignment statement in C requires changing the [:=] operator to [=]. These simple translators are effective only between very similar languages and on uncomplicated source programs. For example, in C there is an equivalent function to the

PASCAL repeat loop. The do...while loop in C can be directly substituted by the automated translator whenever the repeat loop is encountered. Other differences, such as the inability in C to pass parameters by reference, are much more difficult to handle with an automated translator. Differences that are not addressed should be flagged by the translator when such code is encountered. For most functions there is more than one alternate construct. The most efficient alternate construct should be used. Finally, the overall effectiveness of the translation is determined by the level of manual effort necessary to get from the translated code to a compilable, correctly running program.

Two automated translators were reviewed, the Turbo PASCAL-To-QuickC Translator (TPQC) by Microsoft Corporation and PTC by Holistic Technology AB. Both translators are freely available on network bulletin boards. The advertised features of each translator are discussed in the following sections.

a. *TPQC Features*

TPQC converts Turbo PASCAL source code (version 3.0 or earlier) to C source code compilable on either Microsoft's QuickC or C 5.0 Optimizing compilers. TPQC requires that the source code be syntactically correct and that the source code can be compiled and run successfully. The conditions below, if present in the source code, can cause translation errors. The PASCAL source code must be modified before executing TPQC.

- Set Declarations. C does not have set types. To prevent memory allocation errors, remove set declarations from loop statements.

- Nested Procedures. In some cases nested procedures must be modified to prevent forward declaration errors.

- Reserved Words. PASCAL source code procedure and function names that conflict with C reserved words, library-function names, or macros must be changed.

- Forward References of Type. PASCAL pointers of undefined types are assumed to point to PASCAL record types. If this is not the case, the C output code must be modified.

- Case Statements. Case statements used to define variant record structures cannot be translated.

- Compiler Directives. All compiler directives except $C and $I are ignored.

- External Procedures and Functions. External procedures and functions are converted by placing a declaration of the routine in the C program, but no linking occurs. The function must be inserted manually.

- In-Line Machine Code. In-Line machine code is not converted and the C code cannot be run until an appropriate assembly-language function is written.

- Overlays. Overlays are ignored. [Ref. 17]

b. PTC Features

PTC was developed for use on the UNIX operating system. Recompilation for use on the MS-DOS operating system was necessary. PTC provides a self-test function to determine if the recompilation was successful.

PTC is a generic translator which converts any PASCAL program or code fragment into its equivalent in C. When making decisions on multiple alternate constructs for a PASCAL function, the PTC designers followed an interesting philosophy. Instead of selecting the alternate construct which maximizes the efficiency of the resulting C code, the designers selected the alternate construct which most closely complied with the requirements of the PASCAL standard [Ref. 18]. The resulting C code is less efficient and more difficult to understand. The conditions below, if present in the source code, can cause unexpected results. PTC does not automatically flag potential problems. The conditions below should be reviewed as possible sources of problems in compiling or running the translated code.

- Record Variants. PTC uses a complex formula for determining the size of memory to allocate for variant records. The memory allocated may not be adequate.

- Pointers. A pointer defined recursively (e.g., type ptr = ^ptr) cannot be translated.

- Procedure Scoping Rules. PASCAL scoping rules for nested procedures are ignored. Nested procedures dependant on PASCAL scoping rules must be modified. [Ref. 18]

D. COMPARISON AND SELECTION

Five software translation methodologies were reviewed. Three of the methodologies were considered unsuitable for the case study for reasons cited below. These methodologies were the transformation based maintenance model, attribute grammar technology, and manual re-implementation. The primary methodology selected for use in this case study was inverse transformation. Additionally, the automated source code translators were used on selected portions of the case study.

The transformation based maintenance model (TMM) is a complex methodology that requires a major investment in development time. For small programs the development time of the DAG alone can be expected to exceed the time required to develop the program from scratch. For large programs, capturing the information required from the source code to employ TMM matches the complexity and level of effort required to develop a compiler analyzer, and may not be worth such an effort for one-time use. The major advantage predicted for TMM is the possibility of using abstractions from the DAG developed from one application for other program recovery efforts. [Ref. 14] TMM is not suitable for one time application on relatively small programs such as the case study.

Attribute grammar technology also requires a significant investment in development time. Grammars are required for both the source and target languages. The intermediate language bridging the differences in the two languages is also required. The investment in development time is not the most important drawback, however. Applying attribute grammar technology as a software translation tool yields only translated source code. This methodology does not generate requirements or design documentation as output because neither are required as input. For this case study, the creation of this documentation is essential to support future maintenance efforts.

Automated source code translators translate directly from source code to source code without reference to life cycle documentation. The translation problems, such as those noted previously with the two specific automated translators reviewed, illustrate further disadvantages. Automated source code translators are unsuitable as the primary methodology but are potentially valuable to speed the coding of certain portions of the source code.

The inverse transformation methodology is the only methodology reviewed that supports the evolution of life cycle documentation and permits unrestricted determination of the design development strategy. The inverse transformation methodology uses the SDLC as the model for the software translation. The output of this methodology includes both the translated source code and life cycle documentation to support future maintenance.

The methodology selected for the case study was the inverse transformation methodology. Within the framework of this methodology, automated source code translators were also used on portions of the source code as part of the design strategy. Details of the design strategy employed are in Chapter V.

V. DESIGN STRATEGY AND TRANSLATION APPROACH

A. OVERVIEW

This chapter describes the specific approach taken to develop the design strategy used with the inverse transformation methodology. The design schema selected was structured analysis and design. Step one of the inverse transformation methodology was the creation of the design specification. The structured analysis and design tools used to create the design specification were structure charts, data dictionary, and structured English constructs. The second step in the inverse transformation methodology, development of the requirements statement, was not executed for reasons described in the succeeding section.

The translation approach established the transformation rules and defined the process of translation within the scope of the transformation rules. The case study was divided into three sections for independent development from the design specification. The three sections were screen display and data entry; database management; and print routines. These three sections were then merged to complete the translation.

B. REQUIREMENTS ANALYSIS

The importance of the requirements analysis as a phase of the initial software development life cycle has been previously discussed. The inverse transformation methodology encourages the definition of requirements as a step in the transformation process to support future maintenance. In order to fully re-create the requirements statement, information from the original analysis must be available. In the absence of the original requirements statement and program development personnel, the only source for this information is program users. When even program users are unavailable, as is the situation with the case study, information essential to the accurate re-creation of the requirements statement cannot be obtained. Information determined during the requirements analysis (see Whitten [Ref. 4]) which cannot be obtained from the source code alone are:

- The capabilities and processes of the system in existence at the time the new need was identified.
- The constraints affecting system development such as budgets, regulations, and policies.
- The business objectives of the system to include definitions of the expected performance level and prioritizing the objectives.
- The criteria used to determine the degree of success of the development.

- A general description of the inputs, outputs, and processes needed.

Without the above information, the re-creation of documents in the requirements statement, such as the problem statement and data flow diagrams, would not be accurate. Therefore, to avoid misleading program maintainers the requirements statement is not included in this case study.

C. DESIGN STRATEGY

Structured analysis and design tools described by Page-Jones [Ref. 5] were used to develop the design specification for the case study. The basic task of the inverse transformation methodology is to invert the normal design process by working backwards from the source code to the design specification. Structured analysis and design defines the order in which each tool is created. The inverse transformation methodology reverses that order, which is described as follows. Source code is used to produce structured English. Structured English is used to produce the structure chart. The structure chart is used to produce the data dictionary. Each structured analysis and design tool is discussed in the following sections.

1. Structured English

Structured English [Ref. 4] is a tool that combines plain English with simple structured programming constructs to describe program routines. Structured English is written as short, precise sentences describing data transformations and flow of control. Structured English sentences are composed of imperative English verbs describing action, data dictionary terms as the subject of the action, and reserved words commonly used in structured programming to denote the logical flow of the program. There is no universally accepted, formal dialect for structured English. This is an advantage because it allows the software maintainer to establish the compromise between rigid control and the readability that is right for a specific project. Once that balance is reached, consistency of use is the most important factor to keep in mind. The structured English syntax suggested by Whitten, Bentley, and Ho [Ref. 4] provided the baseline for the dialect used in the case study.

In structured analysis and design, data dictionary entries are used as the subject of structured English sentences. In the inverse transformation methodology, these terms are extracted directly from the source code. In the case study, terms used as the subject in structured English sentences were added to the data dictionary as they were introduced. Although this appears to conflict with the pattern of development described above, it was a logical decision which is discussed in more detail in the section on the data dictionary.

The most significant problem encountered in developing the structured English constructs from the source code is the strong tendency to re-write lines of code into English sentences. The result was inevitably too detailed and programming language specific to be useful to the software maintainer. Routines should be generalized first, then written as structured English. The method used in the case study to achieve a generalized view of program routines was to first write out what each routine did in plain English. The English text was then formalized into structured English. The point of this method was to avoid creating the structured English directly from the source code listing.

2. Structure Chart

Structure charts [Ref. 4] are based on the use of structured programming and design techniques for top-down software development. The overall problem to be solved is first identified, then broken down into a series of smaller problems or steps which solve the problem. These steps are further broken down into a series of more detailed steps, building additional levels of steps until the degree of detail required to code the program is achieved. The structure chart graphically depicts this approach.

Structure charts are made up of modules, module connections, and module communications. Modules are graphically illustrated by a rectangle. Within the rectangle is a phrase describing what action is done by the module. The phrase should be very specific about what the module does, not how it is done. Low level modules represent single program functions while higher level modules represent a group of lower level modules which collectively define some larger task. The symbol that represents connections between modules is the arrow. The direction of the arrow determines which module is the calling module and which is the called, or subroutine module. For example, an arrow pointing from module A to module B represents the occurrence of three events: module A calls on module B; module B accomplishes its function; and control is then returned to module A. Module communications is illustrated by a small arrow with a circle on its blunt end. Communication arrows show what information is being sent between modules, with the direction of the arrow showing which direction the information is flowing.

Structure charts were used in the case study for two reasons. First, a graphical method of depicting program design, including inter-relationships between functions and data flow, is an extremely important element of structured analysis and design. Second, of the most commonly used methods (which include but is not limited to Warnier/Orr diagrams, decision trees, and decision tables), structure charts are the most widely used and familiar to software maintainers [Ref. 4].

For the case study the structure chart was developed from the structured English constructs. This method was chosen for consistency with the design strategy. Additionally, a draft structure chart which excluded specific details of communication between modules was developed during the study of the source code. The development of a draft structure chart is a technique that will improve the software maintainer's understanding of the program. It is recommended, but the maintainer should expect significant changes in the final product.

3. Data Dictionary

The data dictionary records information about data used in the program. Each piece of data is given a name. Each name is associated with specific information about the range of values it may acquire and its physical format. In structured analysis, data dictionary entries are drawn largely from data flow diagrams in the requirements statement. In the inverse translation methodology, both the structure chart and structured English constructs are used as sources for data dictionary entries.

Data dictionary data can be one of two types, composite data or data elements. Composite data is data that can be divided into simpler components. Composite data is defined in the data dictionary as the sum of its components. Components of composite data can be either composite data or data elements. Data elements are data which cannot or should not be subdivided into simpler components. Data elements are defined in terms of the values they may acquire.

In the case study entries were made in the data dictionary as soon as entries were identified during the development of both the structure chart and the structured English constructs. This method greatly sped up the completion of the data dictionary and facilitated the verification of the information contained in the data dictionary.

The data dictionary is essential in understanding the way data is used in a program and in helping the software maintainer keep track of the myriad details of the program. The data dictionary is a vital document that should be updated and maintained as the program changes.

D. TRANSLATION APPROACH

The objective of the case study was the translation into C of source code written in PASCAL using the inverse transformation methodology. The translation approach defines the transformation rules followed and the specific steps expected to accomplish the translation. The transformation rules are:

Rule 1. Structured English will be used to describe program routines.

Rule 2. Structure charts will be used to graphically depict program modularity and module relationships.

Rule 3. A data dictionary will be used to describe all information about program data.

Rule 4. The source code will be divided into three sections of development: screen display/data entry; database management; and print routines.

Rule 5. Each development section will be independently evaluated to determine the best method of programming.

Rule 6. The priorities for determining the best programming method are (from highest to lowest priority): direct reuse of source code modules; use of a software tool to generate code; and manual programming from scratch.

Rule 7. Program coding must accurately reflect the design specification.

Within the scope of the transformation rules, the translation process was developed into a sequence of specific steps. Steps in the translation process are summarized in Figure 4. The purpose and expected results of each step are described in the following sections. The actual results and difficulties in executing each step is described in Chapter VI.

1. Step 1: Develop the Design Specification

See section C of this chapter.

2. Step 2: Evaluate Screen Display/Data Entry Development Section

The original source code for this section was developed using a code generator to produce a skeletal framework. The framework underwent major modifications which profoundly reduced the usefulness of the generated code. This appears to be a duplication of effort for reasons which can only be surmised given the lack of development information available. Possible reasons include:

- The developers may have been unaware of the limitations of the code generator.
- User acceptance of the unmodified displays and data entry processes may have been poor.
- Computer memory limitations or other code optimization needs may have led to the modifications.

STEPS IN THE TRANSLATION APPROACH

1. Develop the Design Specification
2. Evaluate the Screen Display/Data Entry Development Area
3. Program the Screen Display/Data Entry Display Area
   a. Develop a Prototype
   b. Identify Deficiencies
   c. Weigh Deficiencies
   d. Make Programming Decision
   e. Test the Programming Effort
4. Evaluate the Database Management Development Area
5. Program the Database Management Development Area
   a. Develop a Prototype
   b. Develop the Linked List
6. Connect the Database Management and Screen Display/Data Entry Prototypes
   a. Program Routines for a Single Record
   b. Program Routines Involving the Linked List
   c. Test Connection Routines
7. Evaluate the Print Routines Development Area
8. Program the Print Routines Development Area
   a. Review the Source Code Documentation
   b. Develop a Prototype Framework
   c. Perform the Automated Translation
   d. Test the Translated Code
   e. Make any Necessary Modifications
9. Connect the Print Routines Prototype
10. Test the Program
    a. Develop the Test Database
    b. Exercise all Program Functions
    c. Demonstrate Source Code Compilability
    d. Demonstrate the Use of a User Database
    e. Correct Discrepancies
11. Review the Tested Program
    a. Delete Unproductive Code
    b. Review Source Code Format
    c. Review Embedded Comments
12. Ongoing Translation Steps
    a. Revise Design Specifications as Necessary
    b. Develop/Update User and Programmer Manuals

Figure 4 - Steps in the Translation Approach

The best method for coding the screen display was determined to be the code generator used in the original development. The old source code could not be reused because many of the routines were hardware dependant and some routines were originally written using inline assembly language. In some cases it was impossible to determine if a routine was a strictly generated routine or one that had been modified. The lack of comments in the source code made the purpose of some routines difficult to determine.

The code generator, Softcode by the Software Bottling Company, generated code in C as well as PASCAL. The features described for data entry processes closely reflect the requirements of the design specification, and greatly reduced the coding effort for data entry validation. The screen display development feature was simple to use and sped up the normally slow process of coding screen graphics.

3. Step 3: Program the Screen Display/Data Entry Development Section

The following steps were defined to accomplish the programming of the screen display/Data Entry development section.

Step 3A: Develop a Prototype. Create a simple test program, or prototype, to evaluate the code generated by the Softcode software. The code generated should be compilable by the Microsoft C compiler. The code should be clearly understandable, consist of modular routines, and be documented with comments. The code should perform the basic functions required by the design specification for the screen display/data entry development section.

Step 3B: Identify Deficiencies. Compare the functionality of the prototype with the requirements of the design specification. Identify as deficiencies design requirements that are not achieved by the prototype.

Step 3C: Weigh Deficiencies. Compare the programming effort required to make the prototype conform to the design specification with the effort of manual programming. Take into account other factors beside the time required to do the programming. Other factors are difficulty in maintaining the code, added complexity, coupling and cohesion considerations, and efficiency.

Step 3D: Make Programming Decision. Based on the evaluation of the code generation deficiencies, make the decision to either modify the prototype or program the development section manually. Complete the initial programming effort.

Step 3E: Test the Programming Effort. Test the program for conformance with the functionality required by the design specification. For example, numeric fields should not accept non-numeric data entries; fields which display computations based on other fields should be verified for correctness; display only fields should not be modifiable; etc. Correct errors and retest until the program works correctly.

4. Step 4: Evaluate the Database Management Development Section

The original source code used the software package BTRIEVE from Novell to perform database management functions. BTRIEVE is a memory resident program that manipulates the database based on instructions provided by the source program via a function call. The database management evaluation was divided into two separate sections, initial display and selection of database records, and updating the database.

Updating the database required routines for adding, deleting, and updating database records. The use of the BTRIEVE software simplified the routines required for these functions. The best translation method for this section was determined to be direct reuse of original source code routines.

Initial display and selection of database records was managed in the original source code by copying all information about every record into an array which was modified concurrently with modifications to the database. The reason for the lack of consideration of memory limitations is unknown. It is possible that the number of records in the database was expected to remain small. Additionally, special routines were written to manage scrolling and highlighting among records, a departure from the screen display methods used for other screens. The translation method chosen for this section was a combination of use of the Softcode code generator to develop the selection screen and the addition of certain manually programmed routines to enhance the generated code. The amount of information held in memory was reduced to only those fields displayed on the selection screen by use of a linked list. Memory for the linked list was allocated only as required.

5. Step 5: Program the Database Management Development Section

The following steps were defined to accomplish the programming of the database management development section.

Step 5A: Develop a Prototype. Develop a prototype program which uses the BTRIEVE database manager to perform these functions: open and close the database; and add, delete, and update database records. Use the test database created by the original program to verify prototype functionality. Use the file format, database structure, and data names used in the original program.

Step 5B: Develop the Linked List. Write routines to manage a linked list of records from the database. Include only data from each record required for display on the selection screen. Functions required are initialize linked link, and add and delete linked list data. Include the routines with the prototype. Re-test the prototype program.

6. Step 6: Connect Database Management and Screen Display/Data Entry Prototypes

The following steps were defined to program the connection between the prototypes into a combined prototype.

Step 6A: Program Routines for a Single Record. Prototype connection routines for single records establish the data paths between data entry screens and the database. Each prototype used unique data naming conventions to maintain clarity about the status and origin of the data. Write routines which assign data retrieved from the database to the data entry screen and routines which assign data modified or added on the data entry screens to the database.

Step 6B: Program Routines Involving the Linked List. Linked list connection routines include the display of the initialized and updated linked list; updating link list data when the data changes; highlighting a specific record for selection; recognition when a specific record is selected; and managing varying numbers of records in the linked list.

Step 6C: Test Connection Routines. Test for the ability to manage a number of records ranging from zero to more than can be displayed at one time on the selection screen. Test that the ordering of records on the linked list is maintained with the same criteria used by the database manager. Test for the smooth movement of the highlight bar from record to record and accurate selection of the highlighted record.

7. Step 7: Evaluate the Print Routines Development Section

The print routines development section was divided into two functional sections. These two sections were report generation and quick printing. The quick printing section required routines to print pre-defined reports of all or part of the data in a single record. Report selection is dependent on which display screen is currently visible when the print function key is used. The report generation section required routines to create, delete, and print user defined reports.

No off-the-shelf software packages were used in the original program to aid in programming the print routines. Routines specifically coded for the print routines were identifiable in the original program. Direct reuse of the routines was selected as the primary translation method.

Instead of recoding the routines manually, the automated translator TPQC was selected as the means of translation. TPQC was selected over PTC because of insurmountable problems recompiling the Unix based PTC program to run on the MS-DOS operating system.

The use of a general automated code translator raised questions similar to those concerning the use of the more tailored Softcode code generator. The translated C code must be compilable with only minimal additional effort by the maintainer. The functionality of the translated C code must be identical to the functionality of the source code.

8. Step 8: Program the Print Routines Development Section

The following steps were defined to accomplish the programming of the print routines development section.

Step 8A: Review the Source Code Documentation. Document all source code routines thoroughly before doing the translation. The automated translator adds no additional comments. Thorough documentation will aid in verifying the accuracy of the translation.

Step 8B: Develop a Prototype Framework. TPQC requires that the source code be syntactically correct and that the program be functionally complete and executable. Develop the framework of a functionally complete program, including variable declarations and the PASCAL equivalent of the function main. Insert the print routines into the framework.

Step 8C: Perform the Automated Translation. Follow TPQC directions.

Step 8D: Test the Translated Code. Review the translated code. Look for obvious errors in translation and code fragments which were not translated. Make a judgement call on the extent of the obvious errors. If problems are major, consider manually translating print routines. If the use of TPQC is still valid, repair minor problems which would force failure of compilation. Compile the translated code and correct errors.

Step 8E: Make any Necessary Modifications. Strip unneeded code from the prototype framework of the translated code. Add routines from the database management development section prototype to retrieve records from the database. Write routines to exercise the print routines. Test the printing of every pre-defined report and the creation, deletion, and printing of reports using the report generator.

9. Step 9: Connect the Print Routines Prototype

Add print routines to the combined prototype. The combined prototype has function keys programmed with skeleton routines for calling pre-defined reports and the report generator. Insert print routines into the combined prototype and add print routine function calls to the skeleton routines. Test the function calls.

10. Step 10: Test the Program

Acceptance test criteria for the case study is limited to the following requirements:

- Retain as a minimum the level of functionality existing in the original program.
- The translated program must be compilable by the Microsoft C optimizing compiler.
- Current users must be able to utilize existing databases without requiring re-entry of data.

The development of a test database was required to exercise the functionality of the translated program. No user or sponsor test database was provided. Therefore, the test database was developed by the software maintainer, which restricted the effectiveness of the functionality test.

The following steps were defined to test the translated program:

Step 10A: <u>Develop the Test Database</u>. Develop the test database in concert with exercising program functions. Begin with a database with no records. The target size for the test database is twenty records.

Step 10B: <u>Exercise All Program Functions</u>. Add, delete, and modify records. Exercise all function key options available for each display screen. Test the use of keyboard keys not defined as options to check for unexpected results. Note discrepancies.

Step 10C: <u>Demonstrate Source Code Compilability</u>. Compile and link all source code with the Microsoft C optimizing compiler.

Step 10D: <u>Demonstrate the Use of a User Database</u>. Use the sample database provided by the sponsor to demonstrate the use of a user database. Although the sample database is not fully developed for use as a test database, it is acceptable to test file format compatibility with the translated program. Exercise step 10B using this database.

Step 10E: <u>Correct Discrepancies</u>. Make program changes as necessary to correct discrepancies discovered during testing. Re-test the program.

11. Step 11: Review the Tested Program

The purpose of this step is to "clean up" the translated source code. Variables and lines of code that do not affect the execution of the program but serve no purpose seem innocuous. However, unproductive code clouds program understanding and makes future maintenance more difficult. Consistent source code formatting aids readability. Comments embedded in the source code are critical to program maintenance. Comments should explain what the code accomplishes, not a line by line description. The following steps were defined to review the tested program:

Step 11A: Delete Unproductive Code. Delete unused variables, including variable declarations and all references to the unused variables. Delete unused lines of code, including definitions, never called functions, and other stray code. Sections of the program which were modified due to discrepancies discovered during testing are prime sections for seeking unproductive code.

Step 11B: Review Source Code Format. Review source code format for consistency.

Step 11C: Review Embedded Comments. Review embedded comments for its value to the software maintainer. Add additional comments where warranted.

12. Step 12: Ongoing Translation Steps

Ongoing translation steps overlap all other steps, proceeding alongside other steps rather than occupying a specific place in the translation approach. These steps overlap because each step in the translation approach may have some impact on the completion of these ongoing steps.

Step 12A: Revise Design Specifications as Necessary. For design specifications to be helpful to the software maintainer, the specifications must accurately reflect the latest version of the program. Modifications made to the program which affect the design specification must be reflected with identical changes to the design specification. Program modifications affecting design specifications are most likely to occur during prototype testing and acceptance testing.

Step 12B: Develop/Update User and Programmer Manuals. Changes to program functionality, the appearance of display screens, and the purpose of user initiated commands must be reflected in the manuals. Additionally, the reasons for the changes, when appropriate for maintenance, should be included in the Programmer manuals.

VI. CASE STUDY APPLICATION

A. OVERVIEW

The design strategy and translation approach described in the preceding chapter was applied to the case study. The practical application of the case study is intended to test the validity of the approach. Departures from the translation approach during the application of the case study are evaluated. The results of the actual execution of each step and any difficulties encountered are described.

B. TRANSLATION APPROACH APPLICATION

Each step is numbered and titled exactly as in Chapter V.

1. Step 1: Develop the Design Specification.

The development of the design specification required the creation of three documents in the following order: structured English, structure chart, and data dictionary. However, it was more practical to produce the data dictionary first using the software tool Source Print. Source Print read the entire PASCAL source code and created a list of all variable names and where those variables appeared in the source code. From this list the declaration of each variable was located to get the information for the entry to the data dictionary. Creating the data dictionary using an automated tool maximized the similarity of variable names between the original source code and the translated source code. Maintaining the same variable names in the translated source code increased the similarity between the original and translated source code and eliminated the need for a variable cross reference list. Using this technique was possible because C supports variable naming conventions which are very similar to PASCAL, and might not have been possible with certain other language combinations. The original order of development for the design specifications should not be revised. In general, structured English is the first document that should be produced unless special circumstances (as in this case study) apply.

The development of the structured English constructs proved to be much more difficult than anticipated. The maintainer's lack of experience with advanced programming techniques, such as the use of overlays, direct access of computer hardware registers, and the use of inline assembly language, was a large stumbling block. These techniques were heavily used in the original source code, and time constraints became a factor in researching and learning the techniques.

Differences in personal programming style between the maintainer and the original developers were also a factor that was not initially considered. Individuals develop programming styles that are familiar and comfortable and helps develop a habit of consistency. In theory, personal programming habits should not be a factor at the design level of development [Ref. 1], and even in practice may not be a problem for many programmers. However, it was a factor for the maintainer. Personal programming style encompasses a wide range of programming habits, but the concern with this case study was the manner in which the program was organized. The original program was not organized poorly. It was organized consistently and was within the bounds of good structured programming practice. However, the form of the organization was different from the habits developed by the maintainer. Since structured English is just one step removed from the source code, this difference directly impacted the development of the structured English. The maintainer was required to make a choice between following the style of the developer or adjusting the style to something more familiar. Selecting the developer's style has the advantage of reinforcing the similarities between the original and translated programs and the disadvantage of working with a programming style that is foreign to the maintainer. Selecting the maintainer's style has the advantages of familiarity and the disadvantages associated with departing from a strict translation. The decision made was to use the programming style of the maintainer. The general functionality of the case study was well understood by the maintainer, but there was uncertainty at the more detailed level about the advanced programming techniques used. For this reason it was felt that maintaining a familiar programming style would yield more consistent, understandable source code and would not detrimentally affect the overall functionality of the translated program.

The structure chart evolved naturally from the structured English and the draft structure chart created during the initial study of the source code. There were no major difficulties in developing the structure chart.

2. Step 2: Evaluate Screen Display/Data Entry Development Section

The evaluation of the screen display/data entry development section was straightforward. No problems with executing this step were encountered.

3. Step 3: Program the Screen Display/Data Entry Development Section

The code generator was used to produce all data entry screens. The code generator also created data field checking routines to ensure that the user entered only valid data. The code produced by the code generator was excellent. The routines were highly modularized, easily understood, and consistently commented with a clear description of the routine's function.

The generated code included a special routine which allowed the maintainer to test the program without requiring additional coding. The maintainer was able to view all screens and test the data entry features of each field. Errors made by the maintainer in programming the code generator were identified early for correction. Changes were made easily and then the code was re-generated. This step was considered to be a pre-prototype step because the generated code did not evaluate the use of function keys and special keyboard keys required by the design specification. This step did validate the appearance of the display screens and data entry checking routines and should have been included as an independent step within Step 3. This step represents the maintainer's only departure from the steps defined within Step 3.

There were no cases where the generated code incorrectly implemented a design specification. There were, however, three specifications that were beyond the capability of the code generator. A description of how the code generator manages data fields is required to explain the problem.

In general, data fields are defined by the code generator as one of two types, display-only fields and fields that can be modified by the user. Modifiable fields are highlighted when the cursor is placed on that field. Display only fields, which cannot be accessed by the user, are coded in such a way that they were not very accessible to the maintainer. The design specifications required that on one screen the user would highlight the field desired and select various options for action on the highlighted field. The specifications further required that these fields could not be modified by the user. The code generator was unable to produce a field that could be highlighted but not modified. Two options were considered to resolve the problem, modifying the generated code and manually coding the problem screen. Since a significant amount of useful generated code would be discarded if manual coding were done, modification of the generated code was selected as the best option.

The second problem concerned the method used by the code generator to calculate and display information computed from other fields on the screen. The code generator required that the position of the decimal in a number had to be permanently assigned and hard coded into the program. Variable decimal positions were not allowed. Calculations based on decimal numbers were dependant on the pre-defined position of the decimal. The design specifications required that the user be allowed to use numbers with variable decimal positions that could be changed at the discretion of the user. The problem was resolved by adding a new routine to handle decimal number data entry and revising the computation routines of the generated code.

The third problem was the lack of generated routines to manage function key and special keyboard key selection by the user to move between screens and perform special functions. The code generator did provide shell routines to facilitate the manual coding process. The largest manual coding effort for this development section was devoted to writing these routines.

In evaluating code generation deficiencies, the problem sections were not considered significant enough to warrant a decision to program the entire development section manually. All problems were satisfactorily resolved and the resulting code conforms to the design specifications. No major deficiencies were noted during testing.

4. Step 4: Evaluate the Database Management Development Section

A major problem in the database was identified during this step. The BTRIEVE record manager can be used with several programming languages, including both PASCAL and C. Based upon the initial review of the BTRIEVE manual, it appeared to the maintainer that the database created by the original PASCAL program was compatible for use by the translated C program. This was not the case.

There is a fundamental difference in the way strings are stored in the two languages. C requires a terminating null character which identifies the end of the string. PASCAL strings do not have this terminating null character because strings are terminated in a different way. The result is that a string will be one character longer in C than will its counterpart in PASCAL. A string without the terminating null character, such as a string stored by a PASCAL program, can create catastrophic problems in a C program. Since the use of the original database was considered a very important requirement, the means to manage the string problem was investigated during the succeeding programming step.

5. Step 5: Program the Database Management Development Section

Prototype development was conducted based on the decision by the maintainer to use the original, PASCAL created database. Original source code was directly reused by manually recoding the PASCAL routines into C for all major specifications (open and close the database; and add, delete, and update records). Additionally, special routines were created to manipulate strings without the terminating null character. The linked list management routines were created manually because they did not exist in the original program. The linked list routines were included in the prototype and testing was completed satisfactorily.

This approach failed during Step 6. The reasons for the failure are discussed in the next section. Due to the failure, the maintainer decided that it was not practical to use the original PASCAL created database. The next best thing was to use the data stored in the original database to build a new, C compatible database. Step 5 was repeated with one additional step added. A conversion program was written to convert the PASCAL database to its equivalent C compatible database. Only slight revision to the prototype was required to accommodate the converted database and the special string management routines were deleted.

6. Step 6: Connect Database Management and Screen Display/Data Entry Prototypes

After the initial completion of step 5, conversion routines to manage the transfer of data between the database and the data entry screens were begun. As the coding process continued, the maintainer became aware that the conversion routines were taking up the bulk of the coding time and that the amount of code being produced was disproportionately large when compared with the size of the routines that actually used the data. This approach appeared to be inefficient and an alternative was sought.

The maintainer contacted technical support personnel at Novell, the makers of BTRIEVE, for advice. The Novell technical personnel could not provide a better method to streamline the conversion process or reduce the risk to the database. They strongly recommended that the PASCAL created database be converted to the C format before being used by the translated C program. Since the conversion program would only have to be run once, as part of the installation of the translated code, and no user entered data from the original database would be lost, the maintainer made the decision to convert the database.

The database conversion program proved relatively simple to build. In retrospect, the database conversion was the better of the two options. No other significant departures from the planned steps were required for Step 6.

7. Step 7: Evaluate the Print Routines Development Section

The evaluation of the print routines development section was completed with no significant problems.

8. Step 8: Program the Print Routines Development Section

Two PASCAL programs were developed, one for quick print routines and the second for the report generator, from the original source code. PASCAL programs were necessary in order to use the automated translators, which required as input an executable PASCAL program.

A prototype framework was built around the quick print routines and an executable PASCAL program was successfully developed. Numerous problems evolved in attempting to translate the PASCAL program to C using TPQC. Minor idiosyncracies, legal in PASCAL but confusing to TPQC, were changed to accommodate TPQC and translation was attempted several times. TPQC continued to flag sections as unacceptable which were legal and compilable in PASCAL. Most frustrating was the fact that the translation process aborted following the identification of each translation error. There was no way to tell how many total errors would have to be corrected. Error messages were sparse and left the maintainer guessing as to what the problem might be. Due to these problems and fading confidence in the ability of TPQC to produce acceptable C code, the use of TPQC was abandoned for the quick print routines program. The PASCAL code was reused by direct manual re-coding, which presented no difficulties.

TPQC did not get a second chance with the report generator routines. The total size of the routines, not counting the framework required to make it a complete program, exceeded the 64 kilobytes program size limit required by Turbo PASCAL 3.0. An attempt was made to compile the code using version 4.0, but basic differences in the design of the two versions (primarily the change from include files to the use of units) made this option infeasible. The source code for the report generator used heavily nested procedures, assembly language, and frequent calls to hardware registers. Manually recoding the original code was considered beyond the experience of the maintainer. The maintainer had a good understanding of the overall functionality of the report generator, and could have coded the report generator manually from scratch, but time did not permit this. The translation of the report generator portion of the print routines development section was determined to be beyond the scope of this thesis.

9. Step 9: Connect the Print Routines Prototype

Skeleton routines were already available to link the quick print routines to the combined prototype. Only minor difficulties were encountered in completing this step.

10. Step 10: Test the Program

In accordance with the testing procedure, the test database was initialized with no records. Records were added to test the record selection process and testing was conducted on function key and special keyboard key use. During the testing of individual fields for correct error checking, a major problem was discovered with the first worksheet screen used for data entry and update of single records. No other screen was affected. Previous testing of this screen had revealed no problems, but only limited data entry into individual fields had been done.

Exercising additional fields on this screen disengaged the function key commands to exit the screen, including the maintainer coded "hotkey" (the ESCAPE key) intended to bypass the problem. The program remained active, and fields could be edited on the screen, but the program would not exit from the screen. Diagnosing this problem took the maintainer several days, but was finally traced to a programmer error that was inadvertently resetting the variable that flagged the exit screen routine. There were no other major problems encountered during the testing phase.

11. Step 11: Review the Tested Program

The Microsoft C compiler included a program to check source code for unproductive code. This program was extremely helpful in eliminating stray code which might have complicated future maintenance. Additionally, Source Print was used to produce a neat, easily readable printed copy of the source code. The use of "pretty printer" programs such as Source Print is recommended.

12. Step 12: Ongoing Translation Steps

Every attempt was made to update the design specifications when practical coding considerations warranted modification of the specifications. In practice, however, this is not an easy task, and the maintainer did not always comply with that step. The maintainer still recommends doing the updates as they happen, but when this is not feasible, the change should be immediately noted in writing so that corrections to the design specifications can be made at a more practical time. The user and programmer manuals began as text files that the maintainer made notes in as the development proceeded. The notes provided a solid base from which to write the formal manuals.

C. CORRECTION OF APPLICATION INCONSISTENCIES

Chapter II, section D described inconsistencies in the original program discovered during the initial review of the case study. Original application inconsistencies which were corrected are listed in the following sections.

1. Screen Movement

All arrow keys provide consistent movement between fields for each data entry screen. The PageUp/PageDown keys are used only for movement between certain screens during the data entry process. When these keys are active, they are displayed as options in the bottom screen menu display.

2. Function Key Use

All function keys which are active for the currently displayed screen are listed in the bottom screen menu display. Each function key is assigned only one function that is consistent throughout the program. When the function key is not displayed in the bottom screen menu, it is not active and nothing will happen if the key is pressed. The <F9> function key was disabled because the change screen color function was eliminated.

3. Report Generator

The translation of the report generator was determined to be beyond the scope of this thesis. A display screen advising the user that this option is not available was provided.

4. Other

The constraints on the number of lines of task description could not be eliminated, but the number of lines allowed were increased. Saving changes to the task worksheet can be accomplished any time upon exiting the task worksheet. It is not required that the user exit the program to save changes to the task worksheet. Additionally, the user will always be asked if changes should be saved.

VII. CONCLUSION

The purpose of this thesis was to investigate software reusability applications and the practical utilization of those applications in the performance of software maintenance. The translation of a functioning program from one high level language to another was selected as the type of software reusability effort to be explored. Five translation methodologies were investigated and the inverse transformation methodology was chosen. A design strategy and translation approach was developed based on the inverse transformation methodology. The translation approach was followed in performing the translation of the case study.

The results of the translation are encouraging. The inverse transformation methodology provided the high level framework necessary to develop the translation approach. From a practical viewpoint, no significant departures from the steps described by the translation approach were necessary to satisfactorily complete the translation. The additional advantage of this methodology was the creation of design specifications for the translated program which can be used in future maintenance efforts. The use of one tool for software reusability, the inverse transformation methodology, created a second tool for software reusability, the design specification.

Finally, the versatility of the inverse transformation methodology, which allows unrestricted determination of the design strategy, permitted the use of additional reusability tools such as code generators. Significant development time was saved despite the documented problems in using these tools.

LIST OF REFERENCES

1. Jones, T. C., "Reusability in Programming: A Survey of the State of the Art", *Tutorial: Software Reusability,* IEEE Computer Society Press, 1987.

2. Horowitz, E., and Munson, J., "An Expansive View of Reusable Software", *Tutorial: Software Reusability,* IEEE Computer Society Press, 1987.

3. Freeman, P., "Reusable Software Engineering: Concepts and Research Directions", *Tutorial: Software Reusability,* IEEE Computer Society Press, 1987.

4. Whitten, J., and Bentley, L., and Ho, J., *Systems Analysis and Design Methods,* Times Mirror/Mosby College Publishing, 1986.

5. Page-Jones, M., *The Practical Guide to Structured Systems Design,* Yourdon Press, 1988.

6. *GAT User's Manual,* EG&G, Washington Analytical Services Center, 1988.

7. Phillips, J., "Creating a Baseline for an Undocumented System - Or What Do You Do With Someone Elses Code?", *The Record of the 1983 Software Maintenance Workshop,* IEEE Computer Society Press, 1984.

8. Fay, S., and Holmes, D., "Help! I Have to Update an Undocumented Program", *The Proceedings of the Conference on Software Maintenance-1985,* IEEE Computer Society Press, 1985.

9. Jensen, K., and Wirth, N., *Pascal User Manual and Report,* Springer-Verlag, 1974.

10. Gehani, N., *C: An Advanced Introduction,* Bell Telephone Laboratories, 1988.

11. Kernighan, B., and Ritchie, D., *The C Programming Language,* Prentice-Hall, 1978.

12. Feuer, A., and Gehani, N., "A Comparison of the Programming Languages C and Pascal", *Comparing and Assessing Programming Languages Ada, C, and Pascal*, Prentice-Hall, 1984.

13. Sneed, H., and Jandrasics, G., "Inverse Transformation of Software from Code to Specification", *Proceedings, IEEE Conference on Software Maintenance*, IEEE Computer Society Press, 1988.

14. Arango, G., and Baxter, I., and Freeman, P., and Pidgeon, C., "TMM: Software Maintenance by Transformation", *Tutorial: Software Reusability*, IEEE Computer Society Press, 1987.

15. Marcotty, M., and Ledgard, H., *Programming Language Landscape*, Science Research Associates, 1986.

16. Yellin, D., "Attribute Grammar Inversion and Source-to-Source Translation", *Lecture Notes in Computer Science*, Springer-Verlag, 1988.

17. *Glockenspiel Turbo Pascal-To-QuickC Translator Howto Documentation*, Microsoft Corporation, 1987.

18. Bergsten, P., *PTC Implementation Note*, Holistic Technology AB, 1987.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0412 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Director, DSS Directorate (DRI-S) Defense Systems Management College Fort Belvior, Virginia 22060-5426	1
4.	LCDR Charles Bell 326 Valley Road Etters, Pennsylvania 17319	1
5.	LCDR Rachel Griffin Code CS/gr Naval Postgraduate School Monterey, California 93943-5002	2