Quantifying Software Maintainability on Re-Engineered Translation of FORTRAN to C++ Code

by

Zane Grey Tomlinson, Jr.

A thesis
submitted to the
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Software Engineering

Melbourne, Florida
July 2004

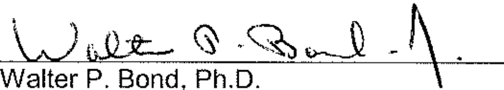
We the undersigned committee
hereby approve the attached thesis

Quantifying Software Maintainability on
Re-Engineered Translation of FORTRAN to C++ Code

by

Zane Grey Tomlinson, Jr.

Rhoda Baggs Koss, Ph.D., Major Advisor
Assistant Professor of Computer Sciences
Program Chair, Computer Information Systems
School of Extended Graduate Studies



Walter P. Bond, Ph.D.
Associate Professor of Computer Sciences
Florida Institute of Technology

Michael D. Shaw, Ph.D.
Associate Professor of Mathematical Sciences
Florida Institute of Technology

William D. Shoaff, Ph.D.
Associate Professor of Computer Sciences
Head, Department of Computer Sciences
Florida Institute of Technology

ABSTRACT

TITLE: Quantifying Software Maintainability on Re-Engineered Translation of FORTRAN to C++ Code

AUTHOR: Zane Grey Tomlinson, Jr.
MAJOR ADVISOR: Rhoda Baggs Koss, Ph.D.

Due to the expanding existence of old software, legacy systems, and obsolete platforms with many industries, software re-engineering has become a widespread methodology that assists engineers and software practitioners with translating inflexible, unsupportable legacy software into maintainable software. Many companies today are investing in a variety of re-engineering techniques such as translation of source code to new code structures and target platforms to ensure future software maintenance can be performed in an efficient and effective manner. With sound re-engineering principles, the application of these techniques leverage the knowledge and previous engineering endeavors to mitigate risks and provide adequate performance to ensure that code attributes retain the functionality of the legacy systems while improving software quality.

In this thesis, an evaluation will be made: What effect does the re-engineering legacy system software have on quality characteristics, with respect to maintainability? The research focuses on determining if a re-engineered methodology of translating FORTRAN to C++ resulting code using an in-house developed translator, can truly re-engineer legacy procedural source code into maintainable object-oriented source code. Based on the metric data and analysis, key measurement results of the empirical data will interpret the translated code to ascertain whether it accurately reflects factors that influence software quality and maintainability. By addressing maintainability and using a set of metrics tailored to assess the criteria, a determination will be made based on the empirical evidence to support the alternative hypothesis that the re-engineered translation of FORTRAN to C++ source code has produced maintainable software. A high-level set of characteristics evaluated in this research include measures quantifying class-related software quality attributes of analyzability, changeability, stability and testability, which include a number of metrics attributes as size, structure, complexity, cohesion and coupling, with emphasis placed on areas of object-oriented characteristics.

The results of this thesis indicate that the re-engineered effort to translate FORTRAN to C++ source code did exhibit maintainable characteristics on the basis that a majority of the metrics examined correlated with high “Maintainability” standards. It is therefore recommended that based on this interpretation of data, opportunities to use the translator in the future for re-engineering efforts should be retained and implemented.

Table of Contents

List of Figures vi

List of Tables vii

Acknowledgement ix

CHAPTER 1 1
Introduction 1
1.1 Overview 1
1.2 Objective of the Thesis 3
1.3 Breadth of Research and Related Work 4
1.4 Purpose 5
1.5 Organization 5

CHAPTER 2 7
Approach 7
2.1 Legacy System Background Data 7
2.2 Thesis Research Methodology 8
2.3 Research Focus 10
2.4 Metrics Analysis 10

CHAPTER 3 11
System Description 11
3.1 FORTRAN to C++ Translator Description 11
3.2 Process Overview 11
3.2.1 Translator Process 12
3.2.2 Restructure Algorithm 12
3.2.3 Slicer Algorithm 12
3.2.4 Solution Algorithm 13

CHAPTER 4 14
Metrics Identification 14
4.1 Overview 14
4.2 Structure 14
4.3 ISO Model 9126 15
4.4 Object-Oriented C++ Characteristics 16
4.5 Metric Identification 17
4.5.1 Cyclomatic Complexity 17
4.5.2 Number of Label References 18
4.5.3 Number of Exits 19
4.5.4 Number of Goto Statements 19
4.5.5 Depth of Inheritance (DIT) 20
4.5.6 Number of Children (NOC) 20
4.5.7 Weighted Methods per Class (WMC) 21

4.5.8 Response for a Class (RFC) 22
4.5.9 Lack of Cohesion in Methods (LCOM) 22
4.5.10 Coupling Between Object Classes (CBO) 23

CHAPTER 5 25
Metric Generation and Data Collection 25
5.1 Cyclomatic Complexity 26
5.2 Number of Label References 27
5.3 Number of Exits 28
5.4 Number of Goto Statements 29
5.5 Depth of Inheritance (DIT) 30
5.6 Number of Children (NOC) 31
5.7 Weighted Methods per Class (WMC) 31
5.8 Response for a Class (RFC) 32
5.9 Lack of Cohesion in Methods (LCOM) 32
5.10 Coupling Between Object Classes (CBO) 33

CHAPTER 6 34
Interpretation of Results 34
6.1 Summary of CSCI Metric Data Results 34
6.2 Interpretation of the Six CSCI Maintainability Characteristics 37
6.2.1 CSCI Maintainability General Results 38
6.3 Overall Synthesis of Maintainability of the Translated C++ Code 39

CHAPTER 7 40
Conclusions 40
7.1 Translator's Effectiveness with Providing Maintainable Object-Oriented Code 40
7.2 Future Work 41

List of References 42

APPENDIX A 45
Metric Measurement Data per CSCI 45
Class Data per CSCI 48

List of Figures

Figure 2-1 Analysis and Evaluation Methodology 9

Figure 5-1 CSCI Average Complexity Data 26
Figure 5-2 CSCI Number of Label Reference Data 26
Figure 5-3 Number of Exits Data per CSCI 27
Figure 5-4 Average Complexity Data per CSCI 27
Figure 5-5 Number of Exits Data per CSCI 28
Figure 5-6 Percent Change in Number of Exits per CSCI 28
Figure 5-7 Number of Goto Statement Data per CSCI 29
Figure 5-8 Percent Change in Number of Goto Statements per CSCI .. 29
Figure 5-9 Depth of Inheritance Data per CSCI #1-3 30
Figure 5-10 Depth of Inheritance Data per CSCI #4-6 30
Figure 5-11 Number of Children Data per CSCI 31
Figure 5-12 Weighted Methods per Class Data 31
Figure 5-13 Response for a Class Data per CSCI 32
Figure 5-14 Lack of Cohesion in Methods Data per CSCI 32
Figure 5-15 Coupling Between Object Classes Data per CSCI 33

List of Tables

Table 1-1 Problems Associated with the Maintenance of Legacy Systems 1
Table 1-2 Approaches to Maintaining Legacy Systems (Historically) 2
Table 1-3 Classes of Structural Measures 3
Table 1-4 Software Topics Related to this Research and Thesis 5

Table 2-1 Obsolescence Problems Presented by the Cybers 8

Table 4-1 ISO 9126 Model Maintainability Sub-Characteristics 16
Table 4-2 Maintainability Measurement Profile 17
Table 4-3 Complexity Evaluation Criteria 18
Table 4-4 Label Reference Evaluation Criteria 19
Table 4-5 Number of Exits Evaluation Criteria 19
Table 4-6 Number of Goto Statements Evaluation Criteria 20
Table 4-7 Depth of Inheritance Evaluation Criteria 20
Table 4-8 Number of Children Evaluation Criteria 21
Table 4-9 Weighted Methods per Class Evaluation Criteria 21
Table 4-10 Response for a Class Evaluation Criteria 22
Table 4-11 Lack of Cohesion Evaluation Criteria 23
Table 4-12 Coupling Between Object Classes Evaluation Criteria 24

Table 5-1 Legacy Computer Software Configuration Items 25

Table 6-1 Quality Assessment Rating Table for CSCI #1 34
Table 6-2 Quality Assessment Rating Table for CSCI #2 35
Table 6-3 Quality Assessment Rating Table for CSCI #3 35
Table 6-4 Quality Assessment Rating Table for CSCI #4 36
Table 6-5 Quality Assessment Rating Table for CSCI #5 36
Table 6-6 Quality Assessment Rating Table for CSCI #6 37
Table 6-7 Metric Descriptive Statistics for each CSCI #1-6 37
Table 6-8 Overall Synthesis of Maintainability of the Translated C++ Code 39

Table 7-1 Conclusive Statements For Maintainability of Translated CSCIs 40

Table A-1 CSCI Average Complexity Data 45
Table A-2 CSCI Number of Label Reference Data 45
Table A-3 CSCI Number of Exits Data 45
Table A-4 CSCI Number of Goto Statement Data 46
Table A-5 CSCI Depth of Inheritance Data 46
Table A-6 CSCI Number of Children Data 46
Table A-7 CSCI Weighted Methods per Class Data 46

Table A-8 CSCI Response for a Class Data 47
Table A-9 CSCI Lack of Cohesion in Methods Data 47
Table A-10 CSCI Coupling Between Object Classes Data 47
Table A-11 CSCI #1 Class Data 48
Table A-12 CSCI #2 Class Data 48
Table A-13 CSCI #3 Class Data 49
Table A-14 CSCI #4 Class Data 49
Table A-15 CSCI #5 Class Data 49
Table A-16 CSCI #6 Class Data 50
Table A-17 CSCI #1-6 Quality Criteria Rating Composite 50

Acknowledgement

I extend my sincere gratitude and appreciation to many people who made this masters thesis possible. Special thanks are due to my Major Advisor, Rhoda Baggs Koss, Ph.D., whose help, stimulating suggestions and encouragement helped me during the research and writing of this thesis. I would also like to acknowledge, with much appreciation, my examining committee, Walter P. Bond, Ph.D. and Michael D. Shaw, Ph.D. for their valuable suggestions and critique.

Especially, I would like to give my special thanks to my wife, Tami, whose support and patient love enabled me to complete this work.

CHAPTER 1

Introduction

1.1 Overview

Software engineering and the art of re-engineering legacy systems has become a significant factor in today's organizations. With the continual widespread advances in technology, many companies are having to make crucial decisions, such as whether to spend money on sustaining older systems by relying on interim fixes until a feasible time at which to upgrade or replace systems become an essential business objective. Key motivators include cost, schedule and performance, with emphasis on overall reliability and functional correctness. As systems mature, the maintenance of legacy systems provides many notable problems, over and above regular maintenance, which drives cost higher. Examples are included below in Table 1-1.

Item #	Description
1	Technology obsolescence
2	The systems have often been modified several times by different programs
3	Lack of supporting documentation. The modifications are often made over a long period of time, with minimal or no documentation
4	The systems are expensive to maintain and have inherent levels of higher risk based on increasing failures
5	Loss of expertise with maintaining the system

Table 1-1 Problems Associated with the Maintenance of Legacy Systems

When these problems become ever present and the system becomes too expensive or too complicated to maintain, renovation of the technology needs to occur. Various approaches are identified in Table 1-2.

Item #	Description
1	Throw out the old system and buy new commercial off-the-shelf product
2	Design and develop a new system in-house from scratch
3	Perform a make-shift re-engineering effort (combination of legacy and new code)
4	Use software re-engineering principles to convert old systems to new versions, which leverage the past knowledge, cost and time

Table 1-2 Approaches to Maintaining Legacy Systems (Historically)

Of the four approaches listed above, this thesis will focus on the fourth approach and the use of software re-engineering principles, also referenced by (Trifu and Dragos) as software renovation, to effectively leverage previous knowledge, functionality, requirements and technologies. It is hoped this will yield a software solution that advances both object-oriented software restructuring and overall maintainability considerations. A key concept with software re-engineering is the approach provides the basis for changes in software systems without changing the functionality (Sommerville). This has triggered a plethora of research with the objective of leveraging the business value of legacy software systems into supportable environments through re-engineering. There are two essential advantages to re-engineering compared to developing new software. The advantages are reduced risk (problems with development, staffing and specification in new software) and reduced costs (the cost of re-engineering is often significantly less than the costs of developing new software) (Gyllenspetz and Tajti; Sommerville).

In this thesis, a business decision will be made to assess the feasibility of re-engineering existing software code, while maintaining its functionality and improving its software quality attributes and properties. The focus of this thesis will be to apply sound software re-engineering principles by investigating and measuring the maintainability of re-engineered source code translated with a translation tool. The translator tool under investigation converts source code from procedural code (FORTRAN) to object-oriented code (C++). The objective of the research is to assess the re-engineering effort to ensure resolution of the supportability concerns with an obsolete system, with the intention of making the source code more understandable and easier to maintain. Key identification of measurement data (metrics) will provide the basis for assessment and analysis. Detailed investigation of the re-engineered FORTRAN to C++ code investigation will determine if the translator is a feasible tool for translating a procedural language into an object-oriented language by validating whether the software changes increase maintainability and produce a more understandable and modular program.

The measurement and evaluation of internal software attributes has played a major role with the improvement of software quality and overall improvements as described in the Software Engineering Institute's Capability Maturity Model (CMM) (Pritchett). From a measurement perspective, specific software quality attributes must be defined to quantify and accurately represent meaning to a user. However, as Pritchett (117) noted, many traditional measures may not be appropriate for object-oriented software and do not address the structural aspects of code. In addition, others have annotated this concern about object-oriented metrics, noting no widespread agreement on which metrics are of value when assessing object-oriented systems (Fenton).

Therefore, given the lack of suitability of traditional measures for use in assessing object-oriented software, this thesis will base the metrics generation data from proposed newer measures identified by Chidamber and Kemerer (4-19) and validated by Pritchett (121-125), whose results concluded that the following object-oriented metrics were validated as being predictors of fault-prone classes: Depth of Inheritance (DIT), Number of Children (NOC), Weighted Methods per Class (WMC), Response for a Class (RFC), Lack of Cohesion in Methods (LCOM) and Coupling Between Object Classes (CBO). Significant importance is placed upon ensuring these metrics provide the substance and granularity to effectively assess the quality attributes of the code under investigation.

Key use of these metrics will generate an effective measurement of internal product attributes by measuring the structural properties of the software. Fenton and Pfleeger (280-319) describe several distinct classes of structural measures, control-flow, information and data-flow and data structure summarized in Table 1-3. It is widely believed that well-designed software is characterized by desirable internal structure attributes and the measurements of these attributes may provide important indicators of key external attributes, such as maintainability, testability, re-usability and even reliability (Fenton and Pfleeger).

Item #	Description
1	Control Flow – addresses the sequence in which instructions are executed in a program
2	Data Flow – follows the trail of a data item as it is created or handled by a program
3	Data Structure – is the organization of the data itself, independent of the program

Table 1-3 Classes of Structural Measures

1.2 Objective of the Thesis

The primary focus of the thesis investigation, from a software re-engineering perspective, of whether an in-house FORTRAN to C++ translator is a feasible tool to utilize during a re-engineering effort. This would require that

obsolete FORTRAN code be re-engineered to object-oriented C++ code to eliminate supportability and maintainability concerns.

The thesis includes determining the effect of translated code on the quality (maintainability) of the resulting code. The investigation will focus on an in-depth evaluation and analysis of the re-engineered source code to determine if it yields maintainable source code.

Based on this representation, this thesis will evaluate measurement data and interpret the results to determine if the code accurately reflects factors that influence the software quality attributes supporting maintainability. The high level characteristics used are analyzability, changeability, stability and testability. The measurement objective will provide the data required to ascertain if the re-engineered code is maintainable based on the resultant data. The null and alternative hypotheses are listed below.

Null Hypothesis: The re-engineered translation of FORTRAN to C++ source code has produced low software maintainability characteristics and therefore should not be used to address the supportability issues of the legacy system until improvements are made in the translator to yield higher software maintainability results.

Alternative Hypothesis: The re-engineered translation of FORTRAN to C++ source code has produced high software maintainability characteristics and therefore is recommended that the translation efforts proceed to mitigate the supportability problems of the legacy system.

1.3 Breadth of Research and Related Work

A key objective of software engineering is to improve the quality of the software, while validating from a measurement perspective; specific software product attributes that meet the customer and user needs. A number of researchers and practitioners have continued to evolve software engineering, focusing on guiding re-engineering processes with the main objective of achieving target quality software attributes and transforming unsupportable code to maintainable, cost effective code solutions. There continues to be an increase in industry focusing on new techniques and tools to enhance legacy system's software and minimize rework and cost. The related work is referenced throughout the thesis and spans a variety of topics as shown in Table 1-4.

Description	References
Object-Oriented Software Engineering	Berard; Chidamber, Darcy and Kemerer; Chidamber and Kemerer; Trifu and Dragos; Whittaker; Eliens; Fenton and Pfleeger; Preiss; Pressman; Sommerville; Whittaker
Software Maintainability Metrics	Daly et al.; Martin; Morris; Pritchett; Rosenberg; Welker and Oman; International Standards Organization; De Marco; Fenton and Pfleeger; Pressman; Sommerville
Software Migration to Object-Oriented Software	Patil, et al.; Zou and Kontogiannis
Software Re-Engineering	Berg; Gyllenspetz and Tajti; The Au; Tahvildari and Kontogiannis; Pressman
FORTRAN and C++ Language	Cary, et al.; Headington; Preiss; Sedgewick; Stroustrup

Table 1-4 Software Topics Related to this Research and Thesis

1.4 Purpose

The purpose of this thesis work is to identify, through data analysis, if the use of an in-house translator tool will be prudent to re-engineer an obsolete system's software to new "target" software architecture. The thesis is aimed at comparing the maintainability of the existing base-lined FORTRAN source code with the translated source C++ code. An analysis on a number of attributes software quality metrics (analyzability, changeability, stability and testability) of the re-engineered translated source code is performed. The outcome of the results will be used to make a recommendation on whether to proceed with the use of the translation tool based on the success in producing maintainable source code for future sustainable activities or pursue other engineering approaches to solve the supportability problems.

1.5 Organization

This section provides an overview of the content of this thesis.

Chapter two describes the approach used to solve the problem. A brief history of the project, methodology, research focus and overview of the metrics software analysis is included.

Chapter three provides a system description of the translation tool, its functionality, data process flows and conceptual overview. The re-engineered

approach is summarized to depict how the translation tool translates FORTRAN procedural code to C++ object-oriented code.

Chapter four identifies the metrics identification and generation used for analysis and provides insight into the validity and scope of each metric specific to measurable software quality attributes.

Chapter five identifies the metrics generation and the collected data and analysis with descriptions and details regarding metric and quantitative assessments.

Chapter six provides the interpretation of results through comparative analysis of software quality metric maintainability attributes (analyzability, changeability, stability and testability) of the re-engineered translated source.

Chapter seven includes the conclusions drawn from the present work and suggestions for future work.

CHAPTER 2

Approach

2.1 Legacy System Background Data

Cyber 860 computers were introduced in the mid-1970's. The Air Force currently operates and maintains the Cyber computer systems network at the Cape Canaveral Air Force Station. The Data Processing System function consists of a Cyber 860 mainframe providing the majority of the data processing during pre-launch; launch countdown and post-launch phases of the Eastern Range operations. The processed data for the operations on the Eastern Range are sent to the Range Operations Control Center via a network for utilization by the Range Safety/Range Control Subsystem (RS/RCS) processors. The Cyber 860 mainframes are water-cooled and are utilized seven days per week on a 24-hour per day basis. The data processed by the Cyber mainframes are stored on a bank of 13 Cyber 885-12 disk drives with each drive equipped with two spindles that turn a disk for data storage.

An outside vendor currently supports the Cyber hardware system. After fiscal year 2006, support could potentially be impacted by the retirement of some of the key personnel that currently support the Cyber. The Cyber should be maintainable for a few years after fiscal year 2006, but the costs could rise significantly, due to loss of expertise and degradation of the Cyber System and the facility systems where the system resides.

While the software applications that run on these computers have matured over the years, the cost to maintain the hardware has been increasing based on maintenance and repair data. There are several factors complicating maintenance of the current system, most of which can be attributed to system obsolescence or quickly approaching obsolescence. An additional impact of continued utilization of Cyber programs is in Table 2-1.

Item #	Description
1	New systems will require programs that translate data products used by the Cyber programs.
2	The Cybers are difficult to network and require proprietary or special interfaces for all input/output requirements.
3	The Cyber Networking Operating System (NOS) has been locally customized for security and other requirements, making an upgrade extremely risky.
4	The Cyber hardware requires high annual maintenance costs.

Table 2-1 Obsolescence Problems Presented by the Cybers

2.2 Thesis Research Methodology

To meet the objectives in the thesis, a case study is used as the research technique where key factors are identified that affect the outcome of an activity. The thesis follows a tailored scientific method to design, collect measurements, analyze and interpret data. Figure 2-1 depicts the analysis and methodology.

The steps identified are as follows:

- Understand the purpose of the measurement (metric) and ensure the validation can be performed or the metric has been validated in previous work.

- Identify the data needed to answer the problem statement, along with the data collection tools and techniques to be used.

- Identify the metrics and measurements used to correlate benchmarks for future analysis and interpretation.

- Gather the data on the translated FORTRAN to C++ code based on maintainability metrics. The translated software will be base-lined to establish an initial benchmark.

- Identify variables, controls and techniques to analyze the data. Analysis is performed to evaluate maintainability criteria of the translated C++ source code.

- Finalize, interpret and present the results. Discovery of results to determine whether the resulting code from the translator is easily maintained and can continue to evolve without referring to the original code.

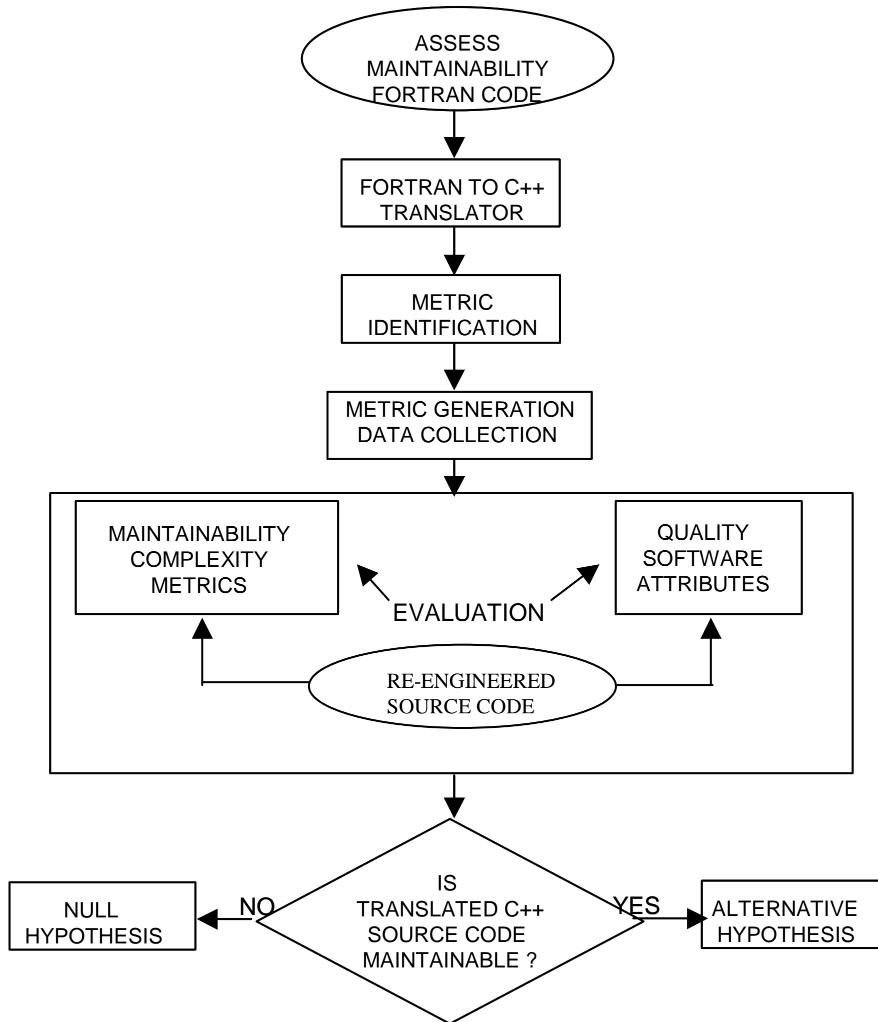


Figure 2-1 Analysis and Evaluation Methodology

2.3 Research Focus

The focus of this research will be to determine whether re-engineering software via automatic translation is an effective approach for meeting defined internal quality attribute standards. The over-riding desired quality objective under evaluation is software quality as it pertains to maintainability. The framework used in this thesis for identifying a methodology for assessing various software quality attributes was tailored from two sources, ISO 9126 model and object-oriented metrics applications (Pritchett and Chidamber, et al.). Based on this tailored framework, a Maintainability Measurement Profile Table is presented in Chapter 4 for identifying key software quality attributes and software structural attributes. This framework will be used during the analysis to help identify if the maintainability is improved significantly after the re-engineering approach and if it meets industry standards associated with object-oriented code.

2.4 Metrics Analysis

Metrics analysis will provide a useful mechanism for assessing the maintainability of the translated code and is the basis for the recommendation on whether to continue to utilize the translator to re-engineer legacy systems to solve supportability problems.

The thesis will look at various software metric measurements for specific properties associated with the translated source code to predict characteristics that have been measured. One example is the use of quantitative complexity metrics, which play an important role with evaluating the effectiveness of re-engineering software systems and the complexity of an entity of a system. To further evaluate this objective, a suite of traditional and object-oriented metrics will be used to measure areas regarding the object structure that reflect complexity of classes and methods and their interactions of class entities. Metrics chosen will support the facilitation of evaluation and fulfill the resolution of the thesis objective for making decisions about whether the software re-engineering effort using the translator will produce maintainable source code. All metrics are listed and further explained in Chapter 4.

CHAPTER 3

System Description

3.1 FORTRAN to C++ Translator Description

Source code translation is used when conversion is made between different programming languages (Gyllenspetz et al.). Translators are tools that convert source code from one programming language to another. If the languages are at the same level of abstraction the most common is that the new code is less readable, but if the new language is at a higher level of abstraction and the conversion is successful, the tool can produce a more understandable and modular program (Berg).

ITT Corporation Systems Division is currently completing the development of a FORTRAN to C++ Translator. The translator is a software tool that translates FORTRAN source code to C++ source code. It is intended for those who wish to convert their FORTRAN code to C++, to yield reproducible results, which are repeatable, verifiable and maintainable.

The tool is a unique translator that takes advantage of a C++ object-oriented environment. Global data blocks are translated into classes and then individual subroutines are assigned to classes, which minimize the number of public variables and methods. The translator was designed to produce code that is optimized for maintainability and compatibility. The power of the translator is in its ability to restructure, slice and optimize code. During the restructuring, the translator replaces unstructured FORTRAN with equivalent structured FORTRAN and eliminates unnecessary Goto statements. The translator also slices FORTRAN modules into smaller modules to reduce module size and complexity. The primary function of the translator is to move applications from a FORTRAN to a maintainable C++ environment by automatic software conversion rather than by expensive, labor intensive re-engineering or re-writing efforts.

3.2 Process Overview

Based on ITT's documentation (ITT System Division), the translator incorporates three key algorithmic components. In the process, each of the global data blocks is translated into classes, which initially contain only data members. The restructure and slicer algorithms are designed so that all modules are reduced

to a low complexity before being assigned to classes by the solution algorithm. A brief description of the translator process and translator algorithms is listed below.

3.2.1 Translator Process

The FORTRAN to C++ translator was developed to reside on a personal computer running Microsoft Windows and developed using Microsoft Visual C++ version 6.0. A procedural FORTRAN Computer Software Configuration Item (CSCI) is translated by running the translator on the CSCI. The translator generates the corresponding C++ source files (cpp and .h) for the CSCI which contains a complete listing of all classes in alphabetical order with hyperlinks to the detailed class descriptions showing all member subroutines with their calling sequences, class data variables, as well as referencing modules and modules referenced. The class descriptions also contain any original FORTRAN comments describing the module. The documentation also provides a calling tree.

After a CSCI has been translated, it can then be compiled and linked on the personal computer. The CSCI is linked with various support libraries that complement the translator by providing FORTRAN intrinsic functions, input/output (I/O) functions and Cyber emulation code.

The final step in the translation process involves manual translation of any formatted I/O statements that were too complex for the translator to interpret. When the translator encounters a formatted I/O statement that cannot be translated, it inserts code to generate a message box so that if the translated program is run, a message box appears indicating that manual translation is required for a particular section of code. The message box specifies the name of the module where manual translation is required and displays the original FORTRAN I/O statement.

3.2.2 Restructure Algorithm

The source code-restructuring algorithm re-writes the non-structured FORTRAN source code into equivalent structured FORTRAN. The translator employs restructuring designed specifically to translate Cyber FORTRAN which employs many extensions to American National Standards Institute (ANSI) standard FORTRAN and local extensions.

3.2.3 Slicer Algorithm

The slicer algorithm locates statements within modules to be sliced. A maximum and a minimum complexity threshold constant for a module are used to control the granularity of module slices. The translator loads the complexity thresholds from a directive file and uses an algorithm to approximate McCabe's complexity algorithm. Any module with a complexity exceeding the maximum threshold is a candidate for slicing. As modules are sliced, the modules created from the slices are appended to the array of modules so that the number of modules grows as modules are sliced. The complexity data presented in this

thesis is averaged over the class and sub-class modules and is applicable not only to how the translator functions but is also used as a data point for the overall structure and complexity of the translated code. The values depicted are averaged values over the entire class structure.

3.2.4 Solution Algorithm

The solution algorithm assigns modules to classes derived from global data blocks, for example, FORTRAN COMMON blocks. The solution algorithm recursively calls itself to ensure that all modules called from the module, which are assignable have been assigned before the module itself is assigned.

CHAPTER 4

Metrics Identification

4.1 Overview

One of the most important steps in the assessment of software quality is the collection of data. The data that is collected has to be useful to develop the findings presented in this thesis. The data collected is based on the legacy procedural FORTRAN code and the translated C++ code, with emphasis on the latter. The analysis provided within this thesis is based on an analysis of code metrics through manual investigation, evaluation and comparison. The interpretation and comparison of identified data provides key insight into resolving the objective of this thesis through the effective validation of the results.

As indicated by DeMarco, "you cannot control what you cannot measure". Software characteristics continue to be promoted by many practitioners as an important consideration with classifying how effective attributes of software impact the desired outcome of the code. As new development and re-engineering technologies become more abundant, a shift in focus from functional properties to non-functional properties is taking on an enhanced interest. Specifically, a non-functional property addresses aspects related to the reliability, compatibility, cost, ease of use, maintenance, maintainability and so forth. Many models have been used in the past, similar to Welker and Oman, which emphasize quantifying software maintainability through prediction variables and maintainability indexes. Evidence suggests through the breadth of research that meaningful metrics should be viewed as a "whole" with other metric relationships and dependency of these relationships based on the scope and depth of the research required.

4.2 Structure

One important focus on the analysis will be to evaluate software structure. Structure plays an important role on how well a product is maintained and measures the structure of the software used to implement the algorithm (Fenton et al.). For example, in this thesis an evaluation of control flow, data flow (hierarchical) and data structure (modular) will be used to extract measurement criteria. The data collected will focus on these three aspects of structural complexity, each playing a crucial role. A brief summary of each is described below:

- Control flow - addresses the sequence in which the instructions are executed in program. This aspect of structure reflects the iterative and looping nature of programs. For example, where lines of code metric counts an instruction just once, control flow measures more visible the fact that an instruction may be executed many times as the program is actually run.

- Data flow - follows the trail of a data item as it is created or handled by a program. Many times, the transactions applied to data are more complex than the instructions that implement them. Data-flow measures depict the behavior of the data as it interacts with the program. As noted by Patil, et al. minimizing data flow interaction complies with principles of information hiding and encapsulation by keeping data flows within boundaries (for example, a class and its associated methods).

- Data structure - is the organization of the data itself, independent of the program. When data elements are arranged as lists, queues, stacks or other well-defined structures, the algorithms for creating, modifying or deleting them are more likely to be well defined. The structure of the data tells us a great deal about the difficulty involved in writing and maintaining programs to handle data and with defining test cases for verifying that the programs are correct.

4.3 ISO 9126 Model

The International Standards Organization published a standard, ISO 9126 Model (International Standards Organization) for measuring software quality that defines quality as a combination of six characteristics. They are: functionality, reliability, usability, efficiency, maintainability and portability. To bind the software characteristic scope of the thesis, the focus will be on the maintainability attribute that according to Pressman (93) “relates to the ease with which a program can be corrected if an error is encountered, adapted if its environment changes or enhanced if the customer desires a change in requirements.” The attribute is comprised of the sub-characteristics, which are analyzability, changeability, stability and testability identified in Table 4-1.

ISO 9126 Model		
Characteristics	Sub-Characteristics	Description
Maintainability	Analyzability	Relates to the effort needed for diagnosis of deficiencies or causes of failures or for identification of parts to be modified.
	Changeability / Reusability	Relates to the effort needed for modification, fault removal or environment change.
	Stability	Relates to the attributes of software that bear on the risk of unexpected effect of modifications.
	Testability	Relates to the attributes that bear on the effort needed for validating modified software.

Table 4-1 ISO 9126 Model Maintainability Sub-Characteristics

4.4 Object-Oriented C++ Characteristics

In addition to the ISO 9126 model, re-engineering efforts today are changing the programming paradigm to take advantage of modern software design principles, *particularly* object-oriented design. An important aspect of this thesis research will also be to consider and evaluate the impact the translation of the FORTRAN to C++ code has from an object-oriented perspective. The software are considered to fully evaluate the measurability of the translated code. As noted in related documentation, Berard emphasized object-oriented technology yields higher productivity and required fewer engineers to accomplish work as compared to traditional software structures. Based on this ascertion, this thesis elects to evaluate the translated code with object-oriented metrics and attributes.

Based on the structural attributes, ISO 9126 Model Maintainability sub-characteristics and object-oriented characteristics an overall maintainability profile chart, Table 4-2, identifies a number of Quality Metric Criteria. The data from the respective Quality Metric Criteria will be compiled quantitatively to derive a resultant output that can be further decomposed and analyzed to yield results regarding the maintainability of the translated code. Table 4-2 provides a summary of the metrics used in this thesis to make a determination if the code is maintainable. The main objective in selecting the criterion is to ensure a well-represented number of metrics can be assessed to validate the application of a maintainability “coding standard”.

The marked boxes identify the applicability of the metric to the respective data and quality criteria.

Metric Data	Quality Criteria	Analyzability	Changeability / Reusability	Stability	Testability
Traditional Metrics Suite	Cyclomatic Complexity	x			x
	Number of Label References	x	x		x
	Number of Exits			x	x
	Number of Goto Statements	x	x	x	x
Object-Oriented Metrics Suite	Depth of Inheritance (DIT)	x	x	x	x
	Number of Children (NOC)	x	x	x	x
	Weighted Methods per Class (WMC)	x	x	x	x
	Response for a Class (RFC)	x	x	x	x
	Lack of Cohesion in Methods (LCOM) per Class	x	x	x	x
	Coupling (AVG_CBO) Between Object Classes per Class	x	x	x	x

Table 4-2 Maintainability Measurement Profile

4.5 Metric Identification

A brief narrative of each Quality Metric Criteria will be discussed in detail with specific criteria for defining and explaining the applicability of the metric on the "Maintainability" of the resultant code. It addition, a brief assertion for the metric will be provided relating to the measure's ability to predict future results for software maintainability, along with quality objective guidelines for each metric respectively.

4.5.1 Cyclomatic Complexity

McCabe's Cyclomatic complexity measures the complexity of code by taking into account the decision structure of the code and application of the algorithms (for example, code that contains loops, if-then-else conditions and so on). Complexity measurements are used within this thesis based on McCabe's Cyclomatic complexity to evaluate complexity factors of the methods in a class for the re-engineered translated C++ code. The data collected provides an average of the overall complexity of the individual methods to fully evaluate the complexity of

the class data. McCabe noted that the program complexity be measured by the cyclomatic number of the program's flow graph (Fenton et al., 293-294). For a program with flow graph 'F', the cyclomatic number is calculated as: $v(F) = e - n + 2$; where F has 'e' arcs and 'n' modes. The cyclomatic number measures the number of linearly independent paths through 'F'. This measure is useful when counting linear independent paths, but it is not at all clear that it defines a complete picture of program complexity. Empirical evidence has suggested that this metric has a strong correlation with the number of faults found during the testing of a software component. The cyclomatic number is a useful indicator of how difficult a program or module will be to test and maintain. In this context, McCabe has suggested that, on the basis of empirical evidence, when v exceeds 10 in any one module, the modularity may be problematic (Fenton et al. 39). This metric is most often used to measure both analyzability and testability attributes.

To calculate, the average complexity is equal to the sum of the complexity methods divided by the total number of application methods.

The descriptive summary above leads to the following assertion: Maintainability of the software decreases as the complexity increases.

Thresholds and guideline criteria for the complexity metric evaluation is shown in Table 4-3.

Threshold Data	Rating Criteria
Complexity < = 10	Good
11 < = Complexity < = 14	Moderate
Complexity > = 15	Poor

Table 4-3 Complexity Evaluation Criteria

4.5.2 Number of Label References

The number of label references provides an overall structure metric based on the modularity of the code. Elimination of label references provides an opportunity to reduce the level of complexity (analyzability) and enhance the structure to expedite improved changeability and testability. The Label Reference metric is used to provide size measurement criteria, emphasizing the percentage of improvement in each class. Since attributes regarding both simplicity and improved structure improve maintainability, this metric is used to measure both analyzability, changeability and testability attributes.

The descriptive summary above leads to the following assertion: The maintainability of the software decreases as the number of label references increase.

Thresholds and guideline criteria for the complexity metric evaluation is shown in Table 4-4.

Threshold Data	Rating Criteria
Improvement > = 50%	Good
25% < = Improvement < = 49%	Moderate
Improvement < = 24%	Poor

Table 4-4 Label Reference Evaluation Criteria

4.5.3 Number of Exits

The number of exits is a measure of the number of exit statements in a class. Exit statements increase the risk of instability because of unwanted effects on program operation. This metric also provides insight into the testability, since each exit point should be verified during code testing activities.

The descriptive summary above leads to the following assertion: The maintainability of the software decreases as the number of exits increase.

Thresholds and guideline criteria for the complexity metric evaluation is shown in Table 4-5.

Threshold Data	Rating Criteria
Improvement > = 50%	Good
25% < = Improvement < = 49%	Moderate
Improvement < = 24%	Poor

Table 4-5 Number of Exits Evaluation Criteria

4.5.4 Number of Goto Statements

The FORTRAN programming language provides Goto statements, which are undesirable, based on standardized concept of control flow. The Goto elimination can improve program structure by translating implicit control structures to explicit control structures like loops and function calls. Similar to label references and number of exit metrics, the Goto metric emphasizes the effectiveness of the structure of the code yielding insights into analyzability, changeability and testability.

The descriptive summary above leads to the following assertion: The maintainability of the software decreases as the number of Goto statements is used in the code.

Thresholds and guideline criteria for the complexity metric evaluation is shown in Table 4-6.

Threshold Data	Rating Criteria
Improvement > = 50%	Good
25% < = Improvement < = 49%	Moderate
Improvement < = 24%	Poor

Table 4-6 Number of Goto Statements Evaluation Criteria

4.5.5 Depth of Inheritance (DIT)

The Depth of Inheritance measure is defined to be the level of the class in an inheritance tree, with the root class being zero. The number of immediate subclasses provide a measure of how many layers of inheritance make up a given class hierarchy. The deeper a class is in the hierarchy, the more likely it will become more complex as it inherits more methods, which in turn could increase risk of unexpected events during modifications. This measure is important because changes to the parent class may impact the descendents, increasing the difficulty of testability and comprehensibility due to deeply nested functions and inheritance layers. Deep trees indicate greater design complexity, but also promote reuse based on inheritance methodologies. Depth of Inheritance is favorable over breadth with respect to reusability and promotes greater method sharing. Chidamber and Kemerer note that a recommendation of Depth of Inheritance of five or less is preferable based on the increase in complexity of deeper hierarchies.

The descriptive summary above leads to the following assertion: Maintainability of the software decreases as the DIT increases.

Thresholds and guideline criteria for the complexity metric evaluation is shown in Table 4-7.

Threshold Data	Rating Criteria
Depth of Inheritance < = 6	Good
7 < = Depth of Inheritance < = 10	Moderate
Depth of Inheritance > = 11	Poor

Table 4-7 Depth of Inheritance Evaluation Criteria

4.5.6 Number of Children (NOC)

The Number of Children is the number of immediate subclasses subordinate to a class in the hierarchy. This measure is important because a large

number of children may indicate a poor design (improper abstraction) or that the sub-classes are too complex and therefore more fault prone. However, the greater number of children also facilitates reuse since inheritance is a form of reuse. If a large Number of Children is present in the class, additional testing of the methods of the class will be required. Based on Chidamber and Kemerer, a high NOC indicates high reuse and an indication of fewer faults within the code.

The descriptive summary above leads to the following assertion: Maintainability of the software decreases as the number of children increase.

Thresholds and guideline criteria for the complexity metric evaluation is shown in Table 4-8.

Threshold Data	Rating Criteria
Number of Children < = 10	Good
11 < = Number of Children < = 30	Moderate
Number of Children > = 31	Poor

Table 4-8 Number of Children Evaluation Criteria

4.5.7 Weighted Methods per Class (WMC)

The Weighted Methods per Class is a count of the methods implemented within a class. This metric is a predictor of the maintainability of the class. A large number of methods provides for a greater potential impact on its derived classes, which in turn may be more likely to be application specific, limiting reusability. A high WMC has been found to lead to more faults and a predictor of stability, time and effort to maintain the class. In addition, Morris suggested a larger number of methods per object class is likely to complicate testing due to the increased object size and complexity. Based on Chidamber and Kemerer, a recommendation is to have an average of 25 methods with an upper threshold of 40 for user intensive classes.

To calculate:

WMC = Number of Methods in a Class

The descriptive summary above leads to the following assertion: The maintainability of the software decreases as the WMC increases.

Thresholds and guideline criteria for the complexity metric evaluation is shown in Table 4-9.

Threshold Data	Rating Criteria
Weighted Methods per Class < = 40	Good
41 < = Weighted Methods per Class < = 60	Moderate
Weighted Methods per Class > = 61	Poor

Table 4-9 Weighted Methods Per Class Evaluation Criteria

4.5.8 Response for a Class (RFC)

The Response for a Class is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. A key attribute of this metric is its ability to assess the complexity of a class through the number of methods and the amount of interaction with other classes. This metric also provides insight into stability. As noted by Martin, most cases that exhibit stability have minimal dependence on other classes and any change has a large impact, which can lead to fewer changes being made by the programmer. Additionally, a large RFC has been found to indicate more faults and have increased complexity. In addition, the testing and debugging of the class becomes complicated given the increased level of understanding on the part of the tester. To calculate the response for class is equal to the number of methods in the class plus the number of remote methods directly called by methods of the class. This measure counts only the first level of calls outside of the class.

Average RFC = number of methods in a class + number of remote methods directly called divided by the total number of RFC per object class.

The descriptive summary above leads to the following assertion: The maintainability of the software decreases as the RFC increases (Chidamber, et al.).

Thresholds and guideline criteria for the complexity metric evaluation is shown in Table 4-10.

Threshold Data	Rating Criteria
Response For a Class <= corresponding range of classes referenced in the class	Good
Response For a Class < = double the corresponding range of classes referenced in the class	Moderate
Response For a Class < = triple the corresponding number of classes referenced in the class	Poor

Table 4-10 Response for a Class Evaluation Criteria

4.5.9 Lack of Cohesion in Methods (LCOM)

Class Cohesion is a measure of the lack of dissimilarity of methods in a class, in essence, that a class performs more than one function. This measure is important as high cohesion indicates good class subdivision, whereas low cohesion increases the complexity and increases the likelihood of faults. Lack of cohesion indicates the classes' operations do not operate on the attributes and,

therefore, is a poor abstraction/poorly designed class. Thus, the class is more likely to contain faults as the operations and attributes have little to do with each other and the implementation of the class may be complex, further impacting changeability, testability and code stability. High cohesion (low LCOM) is desirable, because it promotes encapsulation and indicates high coupling between methods of a class, as seen in well-defined classes. LCOM is counted as the percentage of methods that do not access a specific attribute of a class averaged over all attributes.

The descriptive summary above leads to the following assertion: The maintainability of the software decreases as the LCOM increases.

To calculate, LCOM is counted as the percentage of methods that do not access a specific attribute averaged over all attributes in the class.

Thresholds and guideline criteria for the complexity metric evaluation is shown in Table 4-11.

Threshold Data	Rating Criteria
LCOM < = .5	Good
.51 < = Lack of Cohesion (LCOM) < = .75	Moderate
Lack of Cohesion (LCOM) > = .76	Poor

Table 4-11 Lack of Cohesion Evaluation Criteria

4.5.10 Average Coupling Between Object Classes (CBO)

Average Coupling Between Object Classes is a count of the number of other classes to which a class is coupled. High CBO is undesirable and detrimental to modular design. The goal is to minimize inter-object class coupling to enhance both modularity and encapsulation, thereby reducing complexity, which is a highly desirable property as noted by Zou. The larger the coupling, the higher the sensitivity to changes in other parts of the design and therefore maintenance regarding analyzability and changeability become more difficult. A higher degree of coupling between objects is likely to complicate application maintenance because object interconnections are more complex. The higher the degree of object independence (for example, the more 'uncoupled' objects are from each other) the more likely it is that objects are suitable for reuse within the same applications and the code stability improves. Uncoupled objects should be easier to augment than those with a high degree of 'uses' dependencies, due to the lower degree of interaction. Testability is likely to degrade with a more highly coupled system of objects. Object interaction complexity associated with coupling can lead to increased error generation during development.

The descriptive summary above leads to the following assertion: The maintainability of the software decreases as the average CBO increases.

To calculate, the coupling between object classes is equal to the number of classes to which a class is coupled and equates to the number of public references divided by the immediate subclasses within the class hierarchy.

Average CBO = number of public methods / number of sub-classes

Thresholds and guideline criteria for the complexity metric evaluation is shown in Table 4-12.

Threshold Data	Rating Criteria
Average Coupling Between Object Classes < = the corresponding range of classes referenced in the class	Good
Average Coupling Between Object Classes < = double the corresponding range of classes referenced in the class	Moderate
Average Coupling Between Object Classes < = triple the corresponding range of classes referenced in the class	Poor

Table 4-12 Coupling Between Object Classes Evaluation Criteria

CHAPTER 5

Metric Generation and Data Collection

The data collected is based on the legacy procedural FORTRAN code and the translated C++ code, as identified in Table 5-1. Data collection of code metrics will be extracted based on a set of traditional metrics (for example, cyclomatic complexity, number of levels, number of local variables, direct call functions, number of exists, number of label references and number of Goto statements) and object-oriented metrics, with emphasis on object-oriented techniques (for example, cohesion, coupling and inheritance). The interpretation and comparison of identified data shown in Chapter 5 will provide key insight into resolving the objective of this thesis through the effective validation of the results further discussed in Chapter 6.

CSCIs	Legacy FORTRAN SLOC per CSCI
CSCI #1	1,814
CSCI #2	1,441
CSCI #3	2,009
CSCI #4	876
CSCI #5	1,940
CSCI #6	2,758

Table 5-1 Legacy Computer Software Configuration Items Listing

5.1 Cyclomatic Complexity

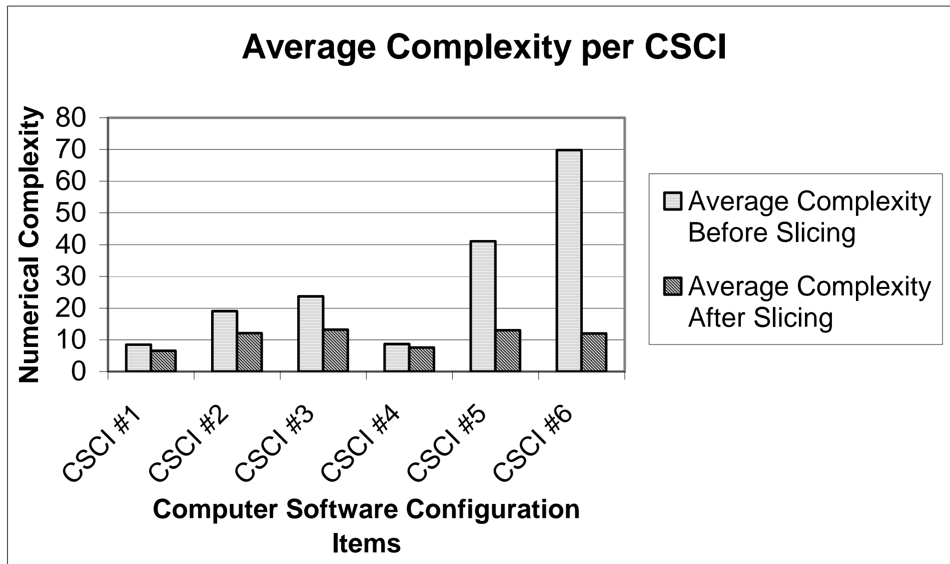


Figure 5-1 Average Complexity Data per CSCI

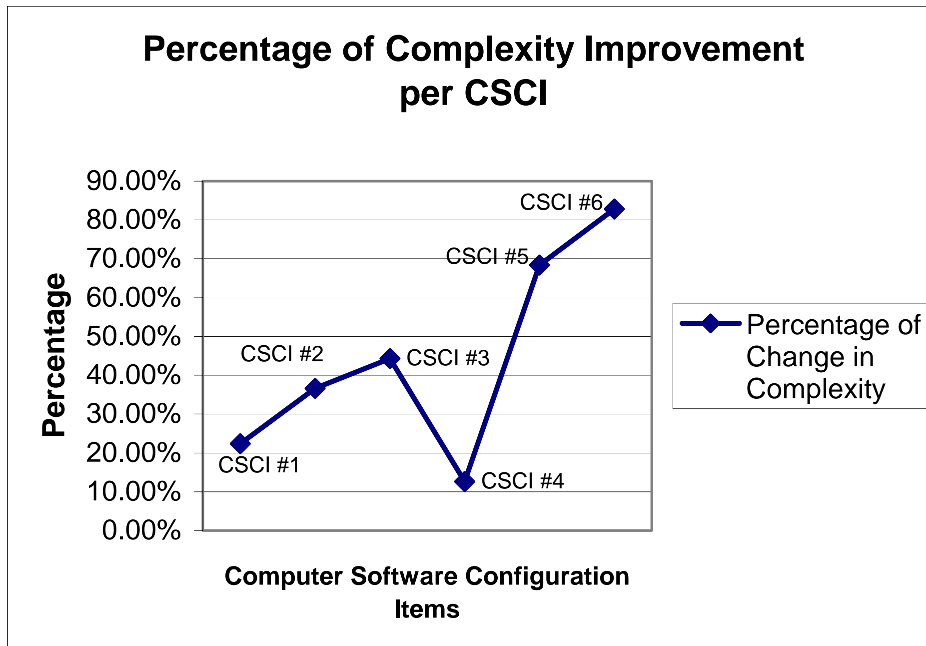


Figure 5-2 Percent Improvement of CSCI Complexity

5.2 Number of Label References

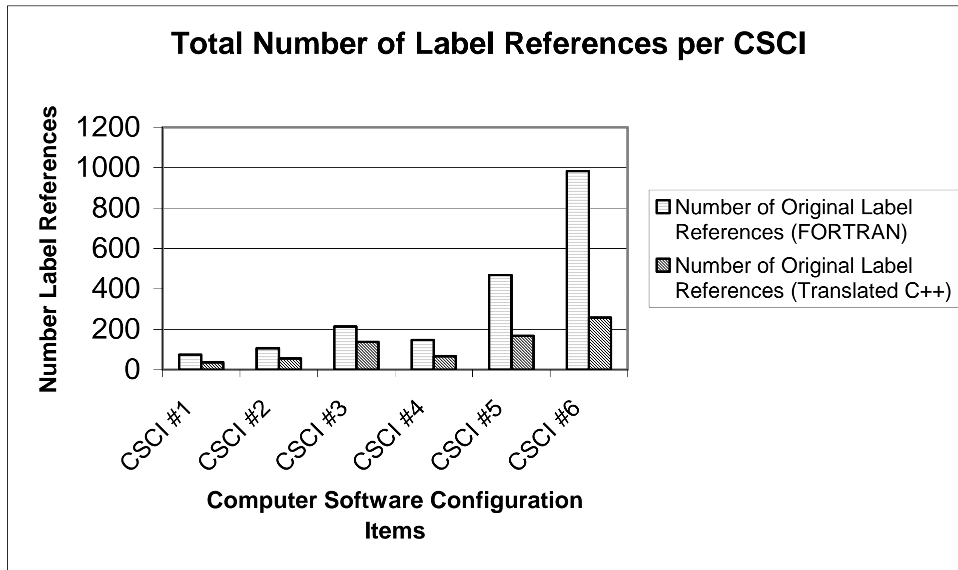


Figure 5-3 Number of Label Reference Data per CSCI

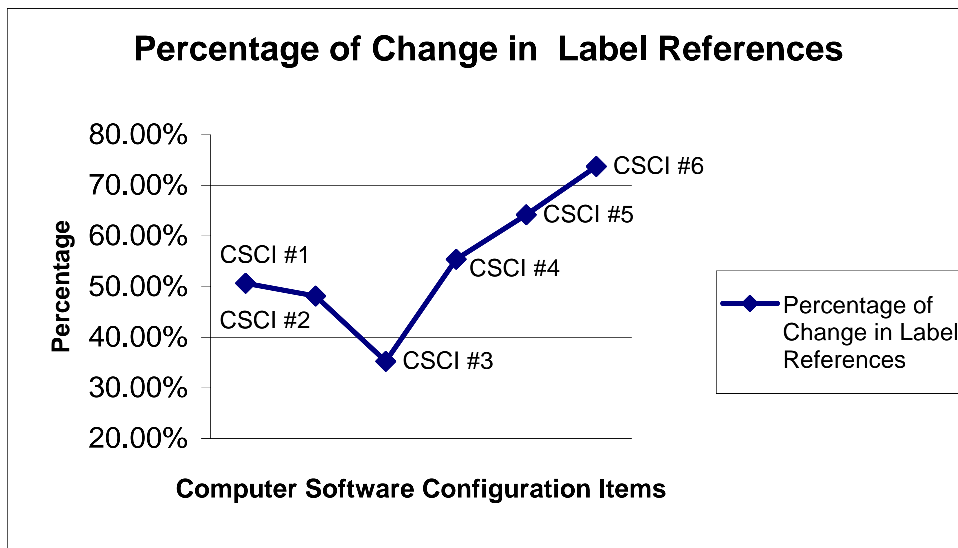


Figure 5-4 Percent Improvement of CSCI Label References

5.3 Number of Exits

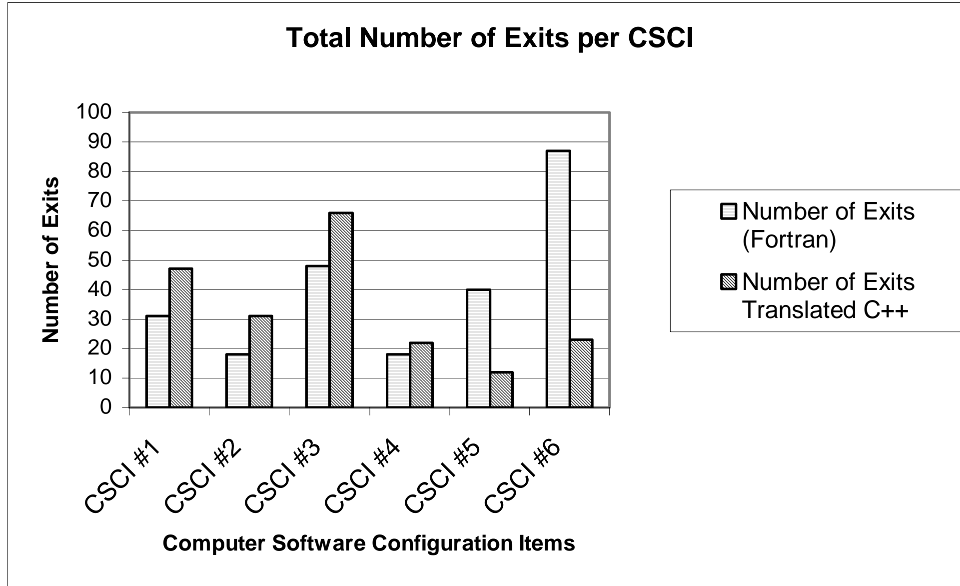


Figure 5-5 Number of Exits Data per CSCI

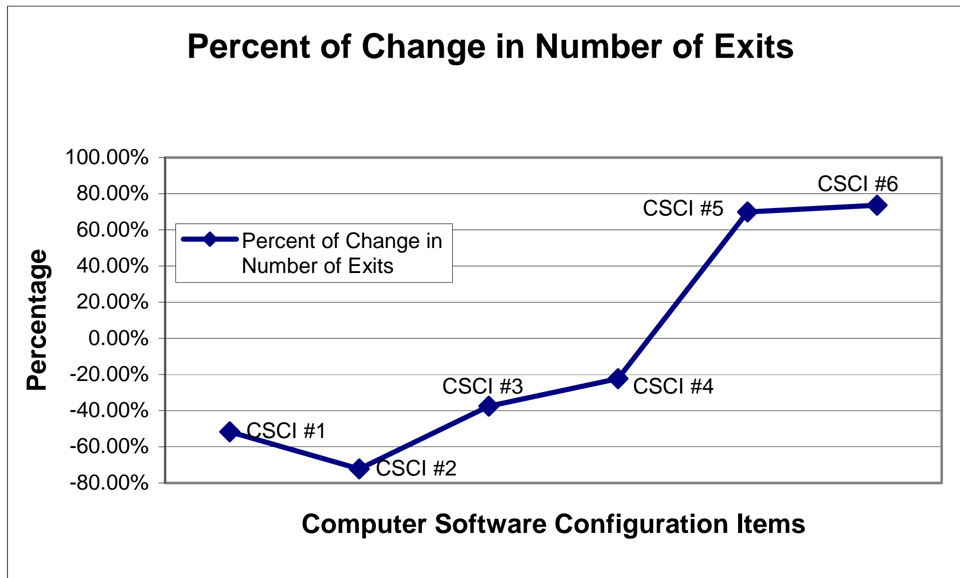


Figure 5-6 Percent Change in Number of Exits per CSCI

5.4 Number of Goto Statements

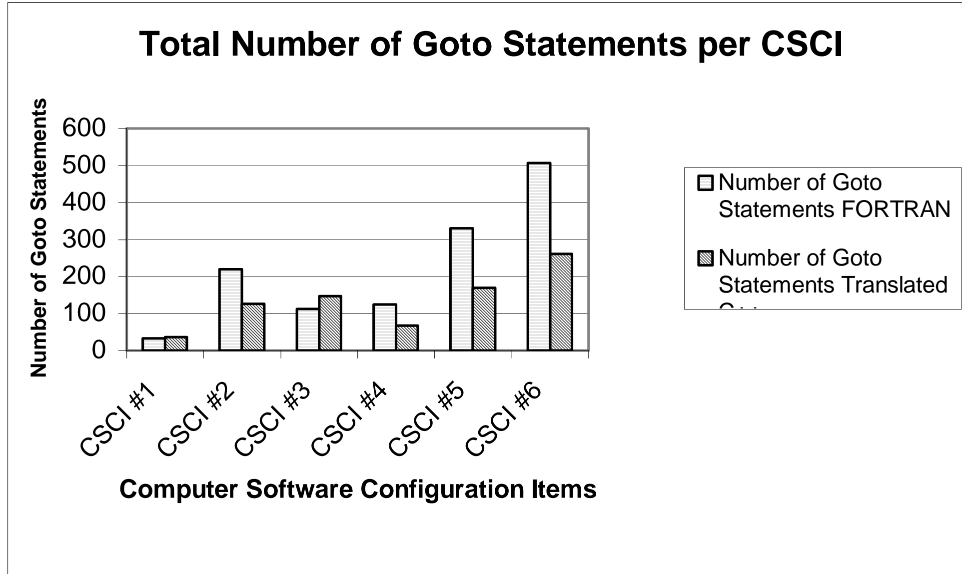


Figure 5-7 Number of Goto Statement Data per CSCI

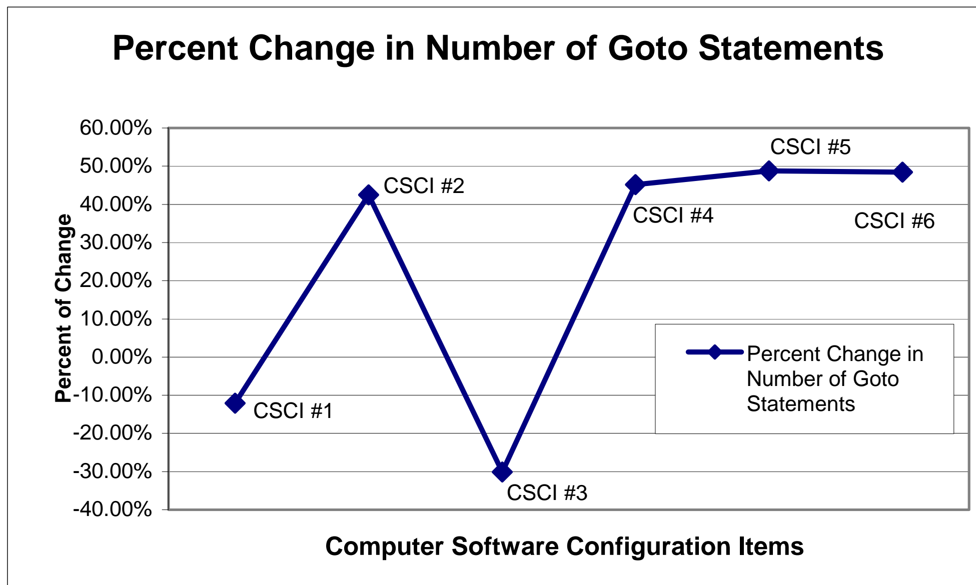


Figure 5-8 Percent Change in Number of Goto Statements per CSCI

5.5 Depth of Inheritance (DIT)

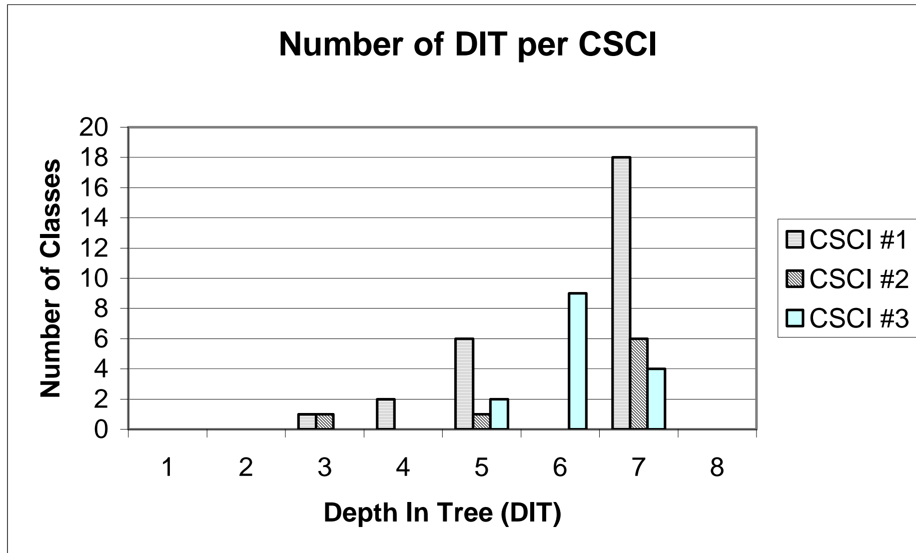


Figure 5-9 Number of Depth of Inheritance Data per CSCI #1-3

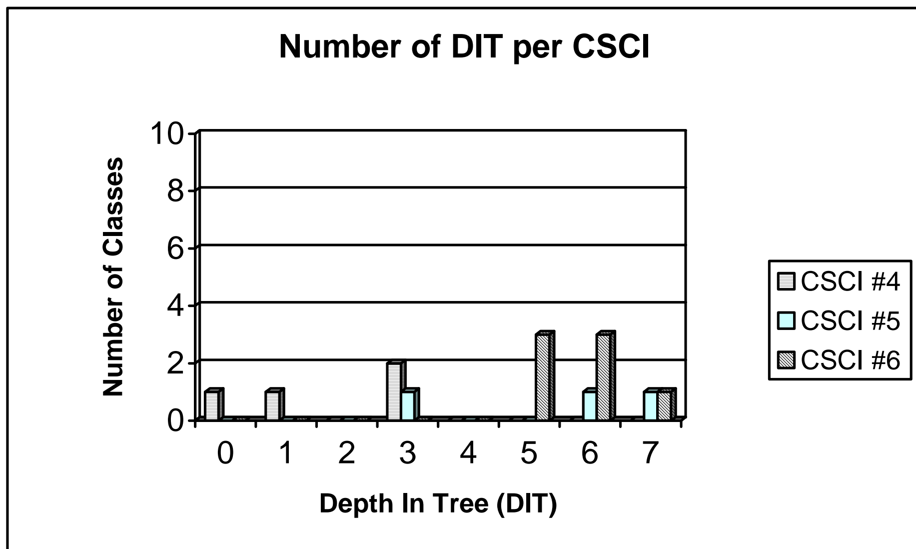


Figure 5-10 Number of Depth of Inheritance Data per CSCI #4-6

5.6 Number of Children (NOC)

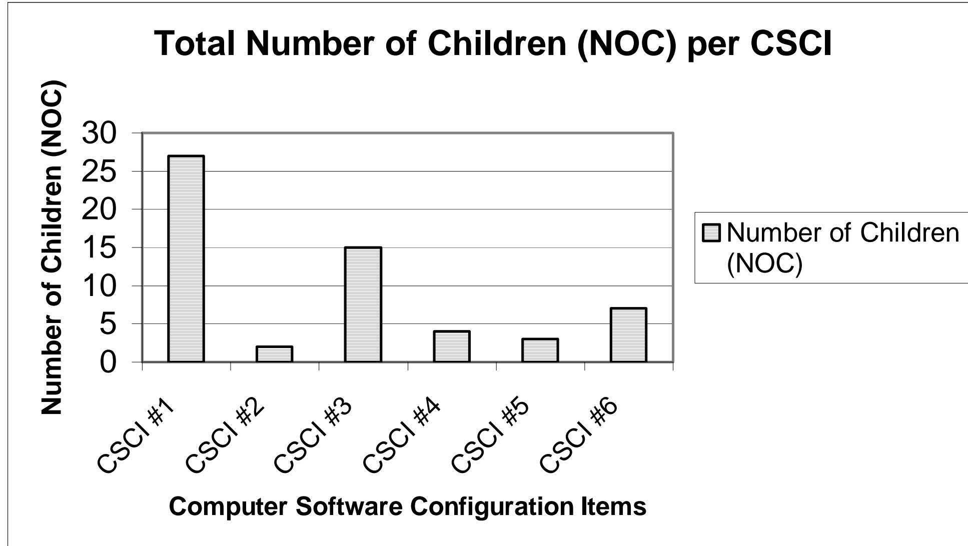


Figure 5-11 Number of Children Data per CSCI

5.7 Weighted Methods per Class (WMC)

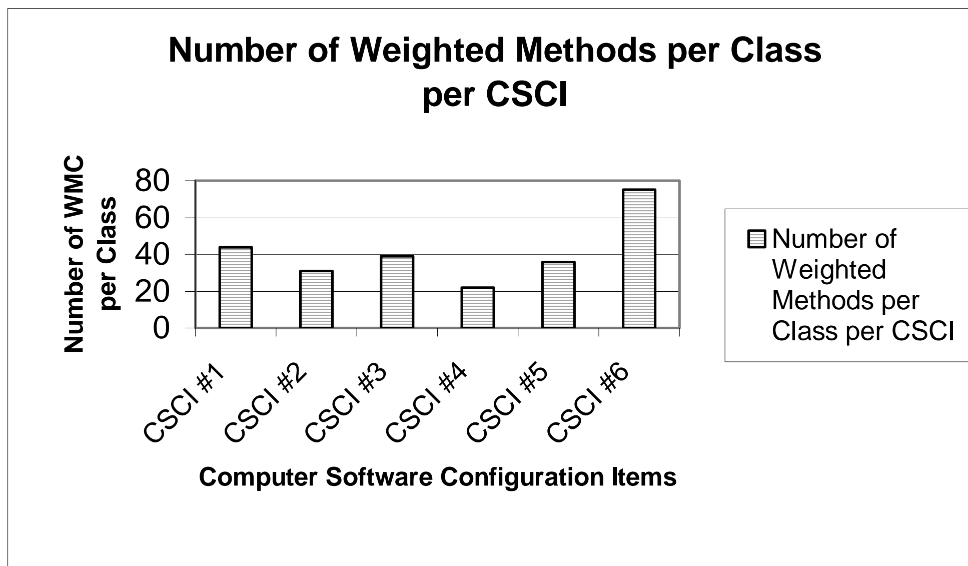


Figure 5-12 CSCI Weighted Methods per Class Data

5.8 Response for a Class (RFC)

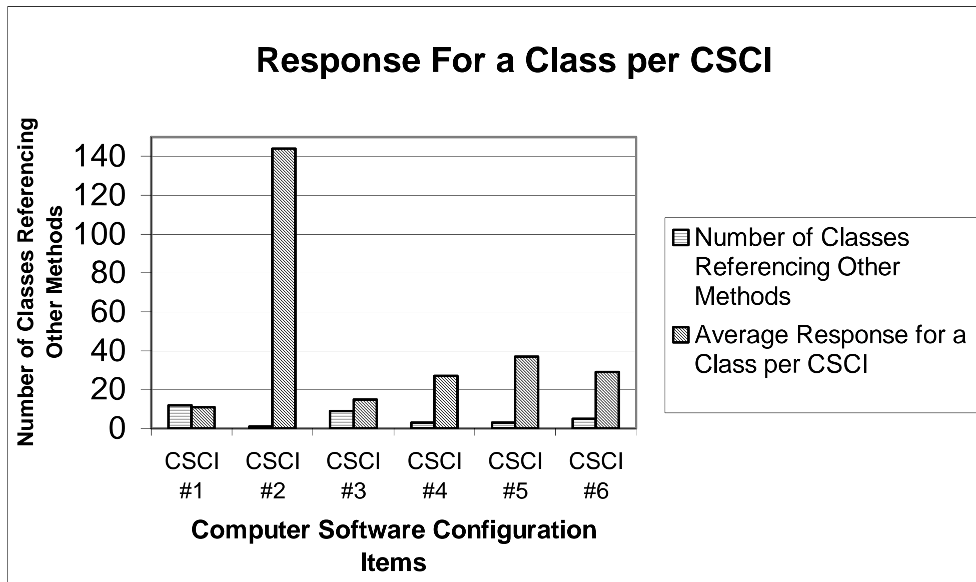


Figure 5-13 Response for a Class Data per CSCI

5.9 Lack of Cohesion in Methods (LCOM)

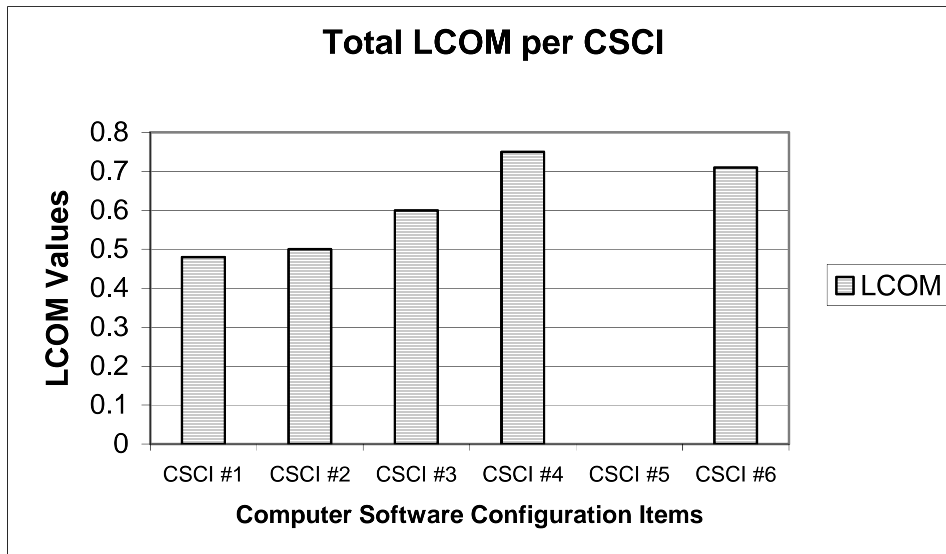


Figure 5-14 Lack of Cohesion in Methods Data per CSCI

5.10 Coupling Between Object Classes (CBO)

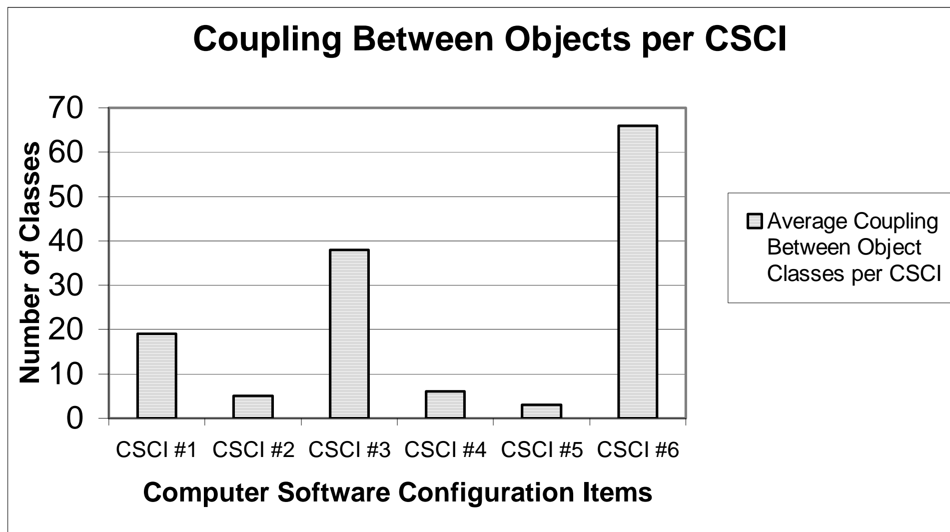


Figure 5-15 Coupling Between Object Classes Data per CSCI

CHAPTER 6

Interpretation of Results

Interpretation of the results is based on the empirical data necessary to validate the result of the thesis hypothesis. To quantify the complexity and quality characteristics, the following tables are derived from each respective CSCI translated code. The data analysis is based on the selected quality characteristics criteria and quality objective guidelines, to make recommendations based on the measured metric characteristic value to assess the maintainability of the translated source code. The six CSCI data are represented on the next several pages.

6.1 Summary of CSCI Metric Data Results

CSCI #1	Quality Metrics For Re-engineered Source Code	Quality Objectives Guidelines	Measured Metric Characteristic Average Value	Ratings - Good - Fair - Poor
DATA	Cyclomatic Complexity	Low	6.6	Good
	Number of Label References	Approx. 50% Improvement	50% Decrease	Good
	Number of Exits	Approx. 50% Improvement	51% Increase	Poor
	Number of Goto Statements	Approx. 50% Improvement	12% Increase	Poor
	Depth of Inheritance (DIT)	Low	6	Good
	Number of Children (NOC)	Low	27	Moderate
	Weighted Methods per Class (WMC)	Low	44	Moderate
	Response for a Class (RFC)	Low	11	Good
	Lack of Cohesion in Methods (LCOM) per Class	LCOM < 1 (Low Value)	.48	Good
	Coupling (AVG_CBO) Between Object Classes per Class	Low	19	Good
OVERALL QUALITY ASSESSMENT RATING				Moderate to Good

Table 6-1 Quality Assessment Rating Table for CSCI #1

CSCI #2	Quality Metrics For Re-engineered Source Code	Quality Objectives Guidelines	Measured Metric Characteristic Average Value	Ratings - Good - Fair - Poor
DATA	Cyclomatic Complexity	Low	12	Moderate
	Number of Label References	Approx. 50% Improvement	48% Decrease	Moderate
	Number of Exits	Approx. 50% Improvement	72% Increase	Poor
	Number of Goto Statements	Approx. 50% Improvement	42% Decrease	Moderate
	Depth of Inheritance (DIT)	Low	6	Good
	Number of Children (NOC)	Low	2	Good
	Weighted Methods per Class (WMC)	Low	31	Good
	Response for a Class (RFC)	Low	144	Poor
	Lack of Cohesion in Methods (LCOM) per Class	LCOM < 1 (Low Value)	.50	Good
	Coupling (AVG_CBO) Between Object Classes per Class	Low	5	Moderate
OVERALL QUALITY ASSESSMENT RATING				Moderate to Good

Table 6-2 Quality Assessment Rating Table for CSCI #2

CSCI #3	Quality Metrics For Re-engineered Source Code	Quality Objectives Guidelines	Measured Metric Characteristic Average Value	Ratings - Good - Fair - Poor
DATA	Cyclomatic Complexity	Low	13.2	Moderate
	Number of Label References	Approx. 50% Improvement	35% Decrease	Moderate
	Number of Exits	Approx. 50% Improvement	37% Increase	Poor
	Number of Goto Statements	Approx. 50% Improvement	30% Increase	Poor
	Depth of Inheritance (DIT)	Low	5	Good
	Number of Children (NOC)	Low	15	Moderate
	Weighted Methods per Class (WMC)	Low	39	Good
	Response for a Class (RFC)	Low	15	Moderate
	Lack of Cohesion in Methods (LCOM) per Class	LCOM < 1 (Low Value)	.60	Moderate
	Coupling (AVG_CBO) Between Object Classes per Class	Low	38	Moderate - Poor
OVERALL QUALITY ASSESSMENT RATING				Moderate

Table 6-3 Quality Assessment Rating Table for CSCI #3

CSCI #4	Quality Metrics For Re-engineered Source Code	Quality Objectives Guidelines	Measured Metric Characteristic Average Value	Ratings - Good - Fair - Poor
DATA	Cyclomatic Complexity	Low	7.6	Good
	Number of Label References	Approx. 50% Improvement	55% Decrease	Good
	Number of Exits	Approx. 50% Improvement	22% Increase	Poor
	Number of Goto Statements	Approx. 50% Improvement	45% Decrease	Moderate
	Depth of Inheritance (DIT)	Low	3	Good
	Number of Children (NOC)	Low	4	Good
	Weighted Methods per Class (WMC)	Low	22	Good
	Response for a Class (RFC)	Low	26	Poor
	Lack of Cohesion in Methods (LCOM) per Class	LCOM < 1 (Low Value)	.75	Moderate
	Coupling (AVG_CBO) Between Object Classes per Class	Low	6	Moderate
OVERALL QUALITY ASSESSMENT RATING				Moderate to Good

Table 6-4 Quality Assessment Rating Table for CSCI #4

CSCI #5	Quality Metrics For Re-engineered Source Code	Quality Objectives Guidelines	Measured Metric Characteristic Average Value	Ratings - Good - Fair - Poor
DATA	Cyclomatic Complexity	Low	13	Moderate
	Number of Label References	Approx. 50% Improvement	64% Decrease	Good
	Number of Exits	Approx. 50% Improvement	70% Decrease	Good
	Number of Goto Statements	Approx. 50% Improvement	49% Decrease	Moderate
	Depth of Inheritance (DIT)	Low	6	Good
	Number of Children (NOC)	Low	3	Good
	Weighted Methods per Class (WMC)	Low	36	Good
	Response for a Class (RFC)	Low	37	Poor
	Lack of Cohesion in Methods (LCOM) per Class	LCOM < 1 (Low Value)	0	Good
	Coupling (AVG_CBO) Between Object Classes per Class	Low	3	Good
OVERALL QUALITY ASSESSMENT RATING				Moderate to Good

Table 6-5 Quality Assessment Rating Table for CSCI #5

CSCI #6	Quality Metrics For Re-engineered Source Code	Quality Objectives Guidelines	Measured Metric Characteristic Average Value	Ratings - Good - Fair - Poor
DATA	Cyclomatic Complexity	Low	12	Moderate
	Number of Label References	Approx. 50% Improvement	73% Decrease	Good
	Number of Exits	Approx. 50% Improvement	73% Decrease	Good
	Number of Goto Statements	Approx. 50% Improvement	48% Decrease	Moderate
	Depth of Inheritance (DIT)	Low	6	Good
	Number of Children (NOC)	Low	7	Good
	Weighted Methods per Class (WMC)	Low	75	Poor
	Response for a Class (RFC)	Low	29	Poor
	Lack of Cohesion in Methods (LCOM) per Class	LCOM < 1 (Low Value)	.71	Moderate
	Coupling (AVG_CBO) Between Object Classes per Class	Low	65	Poor
OVERALL QUALITY ASSESSMENT RATING				Moderate to Good

Table 6-6 Quality Assessment Rating Table for CSCI #6

6.2 Interpretation of the Six CSCI Maintainability Characteristics

Table 6-7 contains the descriptive statistics for each of the metrics considered regarding Points of Central Tendency.

Combined CSCI Measurement Data	Min	Max	Mean	Rating
Complexity	6.6	13.2	10.75	Moderate
# Label References	35% Decrease	74% Decrease	54% Decrease	Good
# Exits	72% Increase	74% Decrease	6% Increase	Poor
# Goto Statements	30% Increase	49% Decrease	25% Decrease	Moderate
DIT	0	7	6	Good
NOC	3	27	9.67	Good
WMC	22	75	41.17	Good
RFC	11.1	144	43.8	Poor
LCOM	.9916	.9979	.50	Good
CBO	5	65.6	22.82	Moderate

Table 6-7 Metric Descriptive Statistics for CSCI 1-6

Table 6-7 shows a central axis for the mass of data to determine the midpoints and distribution. The mean point of central tendency will be used to provide a statistical data point during the overall evaluation rating of each metric.

In addition, an overview of the CSCI software quality metrics and characteristics is discussed below. This criteria and data results compiled in Table A-17, in addition to descriptive statistics shown in Table 6-7 yield a rating (Good, Moderate and Poor) on the maintainability of the translated code, taken as a whole.

6.2.1 CSCI Maintainability General Results

The primary analysis technique used in this thesis is to explore a number of metric measurements and examine focus areas of software quality and object-oriented characteristics. The following is an overview of general results for the six CSCIs under investigation.

First, from a software complexity metric standpoint, the empirical data suggests that the complexity of the translated code significantly improved. The decision structure of the code improved which minimized complexity factors and improved modularity of the code. It appears through the data that complexity improvement will have a positive effect on improving ease of understandability, analyzability and testability. Overall, maintainability attributes were significantly exhibited throughout the translated code.

Second, from a Label Reference, Exit and Goto elimination metric perspective the results were somewhat mixed. There is a significant improvement with the elimination of label reference that reduced the complexity of the code and improved structure. However, the elimination of exits and Goto statements on an average increased. The higher value of exits was expected since the translator provided enhanced modularity with the increase of sub-classes, thereby improving both analyzability and testability. But the lack of decrease in the number of Goto statements showed that while the translator did minimize this use, considerable improvement could be made. The large number of Goto statements can cause instability within the code, impacting overall maintainability. Overall, maintainability attributes were moderately exhibited throughout the translated code.

Third, the empirical results for both Depth of Inheritance and Number of Children appear to show values within acceptable limits, providing insight that these objects can inherit the data and methods and have a potential for code reuse. The modules appeared to be organized into hierarchies that were sensible with advancing ease of management of the application codes. Both DIT and NOC showed a good mix of depth and breadth to leverage both objectives. Overall, maintainability attributes were exhibited throughout the translated code.

Lastly, the results for WMC, RFC, LCOM and CBO are consistent with results in software engineering that higher coupling and lower cohesion are detrimental to maintainability. The data indicates that cohesion was positively emphasized through input and output flow and modularity, providing effective re-organization of the source code (for example, splitting methods and functions). Coupling of the translated code was minimized and provided an assurance that a minimal number of data dependencies between methods improved structure, analyzability and testability. Although all four metrics showed signs of object-oriented influence, the measurements showed moderate object-oriented capability at best with the best improvement in complexity. Overall, maintainability attributes were moderately exhibited throughout the translated code.

6.3 Overall Synthesis of Maintainability of the Translated C++ Code

Combined CSCI CRITERON	Overall Maintainability Rating
Analyzability	Moderate
Changeability	Good
Stability	Moderate
Testability	Moderate
Object-oriented techniques	Moderate to Good
Synthesis (Overall)	Moderate

Table 6-8 Overall Maintainability Synthesis of the Translated C++ Code

CHAPTER 7

Conclusions

7.1 Translator's Effectiveness with Providing Maintainable Object-Oriented Code

This thesis proposed a set of metrics for evaluating the maintainability of a re-engineered translation of FORTRAN to C++ code. It includes the results based on empirical data during the review and evaluation of metric measurements regarding the overall maintainability and software quality attributes. The objective was stated which lead to the resulting alternative hypothesis on how effective the re-engineered effort using the translator was with producing maintainable code.

There is sufficient evidence based on the results of Chapter 5 and Chapter 6, Section 6.2.1, to conclude that the metrics used show the translated code is maintainable. The data exhibited a number of software quality characteristics and taken in the context of "Maintainability", showed a moderate to good correlation of maintainability attributes, resulting in the conclusion that the translated FORTRAN to C++ code meets the intent of supporting maintainable attributes and is indeed maintainable, as identified in the Alternative Hypothesis. It is therefore recommended that the translation efforts proceed to mitigate the supportability problems of the legacy system.

Improved software maintainability is one of the key aspects of the FORTRAN to C++ translator tool. Table 7-1 provides conclusive statements that are forthcoming based on the interpretation of the data.

Item #	Description
1	The translator did produce object-oriented source code minimizing the number of public variables and improved stability.
2	The translator reduced the average module complexity and improved analyzability and changeability by creating smaller, more manageable modules.
3	The translator eliminated unnecessary label references and Goto statements, thus improving code structure, understandability and testability.
4	FORTRAN comments were maintained in the C++ translated source code that assures an adequate level of original readability and analyzability is maintained.

Table 7-1 Conclusive Statements For Maintainability of Translated CSCIs

7.2 Future Work

Future extensions of the work presented in this thesis could focus on two directions. First, further investigation on the use of "other metrics" to support future developmental activities of the translator in order to optimize object-oriented and maintainability attributes. This could extend an improvement in the tailored model and metrics used within this thesis and yield additional design criteria and software quality characteristics during the future enhancement of the translator. Second, additional emphasis on relying on automated tools and techniques for capturing the data and performing the analysis. Commercial-off-the-shelf toolkits are available in the marketplace and could save substantial time and effort during the code analysis and metrics interpretation stages.

List of References

Berard, Edward. “Metrics for Object Oriented Software Engineering.” The Object Agency, Inc. 6 July 2004.

Berg, Martin. “Reverse Engineering PLEX-C code to SDL 10 code.” Lund Institute of Technology, 1999.

Cary, J. R., et al. “Comparison of C++ and FORTRAN 90 for Object-Oriented Scientific Programming.” Computer Physics Communications. Los Alamos National Laboratory. <www.amath.washington.edu/~lf/software/compcap_fgoscioop.html>

Chidamber, Shyame R., David P. Darcy, and Chris F. Kemerer. “Managerail Use of Metrics for Object Oriented Software: An Exploratory Analysis.” University of Pittsburgh, 1997. 28 June 2004. <www.pitt.edu/~ckemerer/fridaysp.htm>

Chidamber S., and Chris F. Kemerer. “Chidamber & Kemerer Object Oreineted Metrics Suite.” 28 June 2004. <www.aivosto.com/project/help/pm-oo-ck.html>

Daly, John, et al. “An Empricial Study Evaluating Depth of Inheritance on the Maintainability of Object-Oriented Software.” Department of Computer Science. University Of Strathclyde, Scotland.

DeMarco, T. Controlling Software Projects: Management, Measurement and Estimation. New York: Yourdon Press, 1982.

Eliens A. Principles of Object-Oriented Software Development. Reading: Addison Wesley, 1994.

Fenton, Norman E., and Shari Lawrence Pfleeger. Software Metrics A Rigorous & Practical Approach. 2nd ed. Boston: PWS Publishing Company, 1997.

Gyllenspetz, J., and Steffan Tajti. “Software Re-engineering From

Function-Oriented to Object-Oriented." Lund University, 2001.

Headington, M. R., and David D. Riley. Data Abstraction and Structures Using C++. Sudbury: Jones and Barlett, 1997.

International Standards Organization. Software quality ISO model 9126.

ITT Systems Division. FORTRAN to C++ Translator. Colorado Springs. 2004.

Martin, Robert. "OO Design Quality Metrics – An Analysis of Dependencies." Illinois, 1994.

McCabe & Associates. Using McCabe Test, Version 7.1. Columbia: McCabe and Associates, 2001.

Morris, Kenneth, L., "Discussion on Object Oriented Metrics." 28 June 2004. <irb.cs.uni-magdeburg.de/sw-eng/us/oop/morris/shtml>

Patil, Prashant, et al. "Migration of Procedural Systems to Network-Centric Platforms." IBM Canada Ltd.

Preiss, B.R. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. New York: John Wiley & Sons, 1999.

Pressman, R. Software Engineering, A Practioner's Approach, 4th ed. New York: McGraw-Hill, 1997.

Pritchett, IV, William W. "An Object-Oriented Metrics Suite for Ada 95." DCS Corporation, 2001.

Rosenberg, Linda H. "Applying and Interpreting Object Oriented Metrics." 28 June 2004. <satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html>

Sedgewick, R. Algorithms in *C*++ Part 5: Graph Algorithms, 3rd ed. Boston: Addison-Wesley, 2002.

Smith, Connie U., and Lloyd G. Williams. Performance Solutions A Practical Guide to Creating Responsive, Scalable Software. Boston: Addison-Wesley, 2002.

Sommerville, Ian. Software Engineering, 6th ed. Reading: Addison Wesley, 2001.

Stroustrup, Bjarne. The C++ Progamming Language 3rd ed. Reading: Addison Wesley, 1997.

Tahvildari, L., and K.Kontogiannis, “A Software Transformation Framework for Quality-Driven Object-Oriented Re-engineering.” IBM Canada Ltd. University of Waterloo.

The Au, Minh. “Java2C Translator.” Monash University, October 1999. 16 September 2003. <www.cssc.monash.edu.au/hons/projects/1999/Minhthe.Au/titlepage.html>

Trifu, Adrian, and Iulian Dragos. “Strategy Based Eliminatin of Design Flaws in Object-Oriented Systems.” Program Structures, Germany.

Welker, Kurt D. & Oman, Paul W. "Software Maintainability Metrics Models in Practice." *Journal of Defense Software Engineering 8*, 11 (November/December 1995): 19-23. 28 October 2003. <www.stsc.hill.af.mil/crosstalk/1995/11/Maintain.asp>

Whittaker, James A. Introduction to Software Engineering. Melbourne: SES Press, 1998.

Zou, Ying, and Kostas Kontogiannis. “A Framework for Migrating Procedural Code to Object-Oriented Platforms.” University of Waterloo.

APPENDIX A

Metric Measurement Data per CSCI

Computer Software Configuration Items (CSCI)	Average Complexity per Module Before Slicing (C++ Code)	Complexity per Module After Slicing (C++ Code)	Percentage of Change in Complexity
CSCI #1	8.5	6.6	22.35%
CSCI #2	19.1	12.1	36.65%
CSCI #3	23.7	13.2	44.30%
CSCI #4	8.7	7.6	12.64%
CSCI #5	41.1	13	68.37%
CSCI #6	69.8	12	82.81%

Table A-1 CSCI Average Complexity Data

Computer Software Configuration Items	Number of Original Label References FORTRAN Code	Number of Label References after Translated C++ Code	Percentage of Change in Number of Label References
CSCI #1	75	37	50.67%
CSCI #2	106	55	48.11%
CSCI #3	213	138	35.21%
CSCI #4	148	66	55.41%
CSCI #5	469	168	64.18%
CSCI #6	983	258	73.75%

Table A-2 CSCI Number of Label Reference Data

Computer Software Configuration Items	Number of Exits FORTRAN Code	Number of Exits Translated C++ Code	Percentage of Change in Number of Exits
CSCI #1	31	47	-51.61%
CSCI #2	18	31	-72.22%
CSCI #3	48	66	-37.50%
CSCI #4	18	22	-22.22%
CSCI #5	40	12	70.00%
CSCI #6	87	23	73.56%

Table A-3 CSCI Number of Exits Data

Computer Software Configuration Items	Number of Original Goto Statements FORTRAN Code	Number of Goto Statements after Transition Translated C++ Code	Percentage of Change in Number of Goto Statements
CSCI #1	33	37	-12.12%
CSCI #2	219	126	42.47%
CSCI #3	113	147	-30.09%
CSCI #4	124	68	45.16%
CSCI #5	330	169	48.79%
CSCI #6	506	261	48.42%

Table A-4 CSCI Number of Goto Statement Data

CSCIs	DIT = 0	DIT = 1	DIT = 2	DIT = 3	DIT = 4	DIT = 5	DIT = 6	DIT = 7
CSCI #1	0	0	1	2	6	0	18	0
CSCI #2	0	0	1	0	1	0	6	0
CSCI #3	0	0	0	0	2	9	4	0
CSCI #4	1	1	0	2	0	0	0	0
CSCI #5	0	0	0	1	0	0	1	1
CSCI #6	0	0	0	0	0	3	3	1

Table A-5 CSCI Depth of Inheritance Data

Computer Software Configuration Items	Number of Children (NOC) (Immediate Subclasses) per CSCI
CSCI #1	27
CSCI #2	2
CSCI #3	15
CSCI #4	4
CSCI #5	3
CSCI #6	7

Table A-6 CSCI Number of Children Data

Computer Software Configuration Items	Number of Weighted Methods per Class (WMC) per CSCI
CSCI #1	44
CSCI #2	31
CSCI #3	39
CSCI #4	22
CSCI #5	36
CSCI #6	75

Table A-7 CSCI Weighted Methods per Class Data

Computer Software Configuration Items	Number of Classes Referencing Other Methods	Total Number Response for a Class per CSCI Sub-Classes	Average Response for Class per CSCI
CSCI #1	12	133	11
CSCI #2	1	144	144
CSCI #3	9	137	15
CSCI #4	3	80	27
CSCI #5	3	110	37
CSCI #6	5	147	29

Table A-8 CSCI Response for a Class Data

Computer Software Configuration Items	LCOM
CSCI #1	.48
CSCI #2	.50
CSCI #3	.60
CSCI #4	.75
CSCI #5	0
CSCI #6	.71

Table A-9 CSCI Lack of Cohesion in Methods Data

Computer Software Configuration Items	Total Number of Public References per Object Classes	Average Coupling Between Object Classes per CSCI
CSCI #1	505	19
CSCI #2	10	5
CSCI #3	574	38
CSCI #4	24	6
CSCI #5	10	3
CSCI #6	459	66

Table A-10 CSCI Coupling Between Object Classes Data

Class Data per CSCI

CSCI #1	Sum # Methods	# Public Methods	# Private Methods	# Referenced Calls	# Referencing Calls	Depth Inheritance	# Variables
1	0	0	0	0	0	3	0
2	1	1	0	0	1	6	12
3	0	0	0	0	0	6	0
4	4	2	2	2	3	6	9
5	2	2	0	0	2	6	6
6	1	1	0	2	1	6	18
7	0	0	0	0	0	6	0
8	0	0	0	0	0	6	0
9	0	0	0	0	0	6	0
10	1	1	0	0	1	6	8
11	1	1	0	0	1	3	5
12	0	0	0	0	0	6	0
13	0	0	0	0	0	6	0
14	0	0	0	0	0	6	0
15	1	1	0	1	1	4	8
16	0	0	0	0	0	6	0
17	4	2	2	1	2	4	7
18	0	0	0	0	0	2	0
19	10	5	5	12	10	4	82
20	12	1	11	15	15	6	51
21	0	0	0	0	0	4	0
22	0	0	0	0	0	6	0
23	0	0	0	0	0	6	0
24	4	1	3	9	5	4	77
25	1	1	0	0	1	4	9
26	0	0	0	0	0	6	0
27	2	1	1	2	2	6	12
Total	44	20	24	44	45	140	304
Average	1.63	0.74	0.89	1.63	1.67	5.19	11.26

Table A-11 CSCI #1 Class Data

CSCI #2 Sub-Classes	Sum Number of Methods	Number of Public Methods	Number of Private Methods	Number of Referenced Calls	Number of Referencing Calls	Depth of Inheritance	Number of Variables
1	0	0	0	0	0	2	0
2	31	1	30	65	48	4	322
Total	31	1	30	65	48	6	322
Average	10.33	0.33	10.00	21.67	16.00	2.00	107.33

Table A-12 CSCI #2 Class Data

CSCI #3 Sub-Classes	Sum Number of Methods	Number of Public Methods	Number of Private Methods	Number of Referenced Calls	Number of Referencing Calls	Depth of Inheritance	Number of Variables
1	1	1	0	1	1	4	1
2	2	1	1	2	2	5	3
3	2	2	0	1	2	6	0
4	0	0	0	0	0	5	0
5	3	3	0	2	3	5	17
6	15	1	14	27	17	6	121
7	3	3	0	1	11	5	11
8	0	0	0	0	0	5	0
9	1	1	0	0	1	6	0
10	0	0	0	0	0	6	0
11	1	1	0	0	1	5	30
12	0	0	0	0	0	4	0
13	11	4	7	15	11	5	98
14	0	0	0	0	0	5	0
15	0	0	0	0	0	5	0
Total	39	17	22	49	49	77	281
Average	2.60	1.13	1.47	3.27	3.27	5.13	18.73

Table A-13 CSCI #3 Class Data

CSCI #4 Sub-Classes	Sum Number Methods per Class	Number of Public Methods	Number of Private Methods	Number of Referenced Calls	Number of Referencing Calls	Depth of Inheritance	Number of Variables
1	2	1	1	1	2	1	7
2	9	6	3	11	17	3	49
3	11	1	10	17	10	3	63
4	0	0	0	0	0	0	
Total	22	8	14	29	29	7	119
Average	5.5	2	3.5	7.25	7.25	1.75	29.75

Table A-14 CSCI #4 Class Data

CSCI #5 Sub-Classes	Sum Number Methods per Class	Number of Public Methods	Number of Private Methods	Number of Referenced Calls	Number of Referencing Calls	Depth of Inheritance	Number of Variables
1	3	1	2	2	3	6	37
2	32	1	31	34	34	7	429
3	1	1	0	0	1	3	13
Total	36	3	33	36	38	16	479
Average	12.00	1.00	11.00	12.00	12.67	5.33	159.67

Table A-15 CSCI #5 Class Data

CSCI #6 Sub-Classes	Sum Number Methods per Class	Number of Public Methods	Number of Private Methods	Number of Referenced Calls	Number of Referencing Calls	Depth of Inheritance	Number of Variables
1	9	3	6	14	9	6	62
2	11	1	10	13	11	5	42
3	40	6	34	45	49	6	219
4	0	0	0	0	0	5	0
5	4	3	1	2	4	5	2
6	11	7	4	12	13	6	28
7	0	0	0	0	0	7	0
Total	75	20	55	86	86	40	353
Average	10.71	2.86	7.86	12.29	12.29	5.71	50.43

Table A-16 CSCI #6 Class Data

Quality Criteria	CC	LR	Exits	Goto	DIT	NOC	WMC	RFC	LCOM	CBO
Analyzability										
CSCI #1	Good			Poor			Mod	Good		
CSCI #2	Mod			Mod			Good	Poor		
CSCI #3	Mod			Poor			Good	Mod		
CSCI #4	Good			Mod			Good	Poor		
CSCI #5	Mod			Mod			Good	Poor		
CSCI #6	Mod			Mod			Poor	Poor		
Avg. Overall Rating	Mod			Mod			Good	Poor		
Changeability										
CSCI #1		Good			Good	Mod	Mod		Good	Good
CSCI #2		Mod			Good	Good	Good		Good	Mod
CSCI #3		Mod			Good	Mod	Good		Mod	Mod
CSCI #4		Good			Good	Good	Good		Mod	Mod
CSCI #5		Good			Good	Good	Good		Good	Good
CSCI #6		Good			Good	Good	Poor		Mod	Poor
Avg. Overall Rating		Good			Good	Good	Good		Good	Mod
Stability										
CSCI #1		Good	Poor	Poor						
CSCI #2		Mod	Poor	Mod						
CSCI #3		Mod	Poor	Poor						
CSCI #4		Good	Poor	Mod						
CSCI #5		Good	Good	Mod						
CSCI #6		Good	Good	Mod						
Avg. Overall Rating		Good	Poor	Mod						
Testability										
CSCI #1	Good		Poor	Poor	Good	Mod		Good		
CSCI #2	Mod		Poor	Mod	Good	Good		Poor		
CSCI #3	Mod		Poor	Poor	Good	Mod		Mod		
CSCI #4	Good		Poor	Mod	Good	Good		Poor		
CSCI #5	Mod		Good	Mod	Good	Good		Poor		
CSCI #6	Mod		Good	Mod	Good	Good		Poor		
Avg. Overall Rating	Mod		Poor	Mod	Good	Good		Poor		

Table A-17 CSCI #1-6 Criteria Rating Composite