AU/ACSC/173/2000-04

AIR COMMAND AND STAFF COLLEGE

AIR UNIVERSITY

THE SOFTWARE MAINTENANCE SPECTRUM: USING MORE THAN JUST NEW TOYS

by

Ricky E. Sward, Maj, USAF

A Research Report Submitted to the Faculty

In Partial Fulfillment of the Graduation Requirements

Advisor: Lt Col Jerry Quenneville

Maxwell Air Force Base, Alabama

April 2000

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burder for this collection of information is estibated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burder to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) 01-04-2000	2. REPORT TYPE Thesis	3. DATES COVERED (FROM - TO) xx-xx-2000 to xx-xx-2000

4. TITLE AND SUBTITLE The Software Maintenance Spectrum: Using More than Just New Toys Unclassified	5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Sward, Rickey E. ;	5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME AND ADDRESS Air Command and Staff College Maxwell AFB, AL36112	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME AND ADDRESS ,	10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APUBLIC RELEASE
,

13. SUPPLEMENTARY NOTES

14. ABSTRACT

As the Air Force enters the 21st century, the software that runs our information systems continues to age and become harder to maintain. Organizations that maintain this legacy software are faced with the challenge of rising software maintenance costs. This paper presents a spectrum of software maintenance options that can be used to reduce the cost of maintenance. The software understanding and programmer unfamiliarity factors from the COCOMO II model are compared to graphically show the effect good software understandability can have on the cost of maintenance. The structure, application clarity, and self-descriptiveness of a software module affect its understandability. The cost of software maintenance is quantified by combining factors from the COCOMO II model and the Software Reengineering Assessment Handbook reengineering decision model that affect understandability. A spectrum of software maintenance options include status quo, redocument, reverse engineer, translate source code, restructure within a paradigm, restructure into a new paradigm, and new acquisition is presented. The benefits of each option are presented in terms of their effect on understandability. Organizations faced with rising maintenance costs should consider the full spectrum of software maintenance options before choosing to replace a software module through new acquisition. By using automated tools, some of the reengineering options present a cost-effective way to reduce the cost of software maintenance.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT Public Release	18. NUMBER OF PAGES 42	19. NAME OF RESPONSIBLE PERSON Fenster, Lynn lfenster@dtic.mil
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER International Area Code Area Code Telephone Number 703767-9007 DSN 427-9007

Disclaimer

The views expressed in this academic research paper are those of the author and do not reflect the official policy or position of the US government or the Department of Defense. In accordance with Air Force Instruction 51-303, it is not copyrighted, but is the property of the United States government.

Contents

List of Illustrations

Page

List of Tables

Page

Preface

I'd like to acknowledge the advice I got on this paper from my faculty research advisor, LtCol Jerry Quenneville. Thanks, Q, for your insights, direction, and inputs. You steered me towards the right blend of analysis, synthesis, theory, and practicality. I enjoyed our meetings…especially the late afternoon ones…

I also need to acknowledge all the support I got from my friends here at ACSC. Thanks, Fritz and Mike for making dinners for me while I put the finishing touches on my paper. Thanks Doug, Chris, and Mary, for letting me bounce my blue sky ideas off of you. Thanks to Seminar 4 for your support…except Todd…no one should finish their paper **that** early. Thanks to Seminar 007 for your support…thanks, Bull, for keeping me on course and on glideslope. Thanks also go to my friends and family that supported me.

Finally, I'd like to dedicate this paper to my Mom and Dad. Thanks for your support. I love you!

Abstract

As the Air Force enters the 21st century, the software that runs our information systems continues to age and become harder to maintain. Organizations that maintain this legacy software are faced with the challenge of rising software maintenance costs. This paper presents a spectrum of software maintenance options that can be used to reduce the cost of maintenance. The software understanding and programmer unfamiliarity factors from the COCOMO II model are compared to graphically show the effect good software understandability can have on the cost of maintenance. The structure, application clarity, and self-descriptiveness of a software module affect its understandability. The cost of software maintenance is quantified by combining factors from the COCOMO II model and the Software Reengineering Assessment Handbook reengineering decision model that affect understandability. A spectrum of software maintenance options include *status quo, redocument, reverse engineer, translate source code, restructure within a paradigm, restructure into a new paradigm,* and *new acquisition* is presented. The benefits of each option are presented in terms of their effect on understandability. Organizations faced with rising maintenance costs should consider the full spectrum of software maintenance options before choosing to replace a software module through new acquisition. By using automated tools, some of the reengineering options present a cost-effective way to reduce the cost of software maintenance.

Part 1

Introduction

As we enter the 21st century, one thing is certain: aerospace power will become even more reliant on information as the cornerstone for every military operation we are ordered to undertake.

—Lt Gen William Donahue, HQ USAF/SC

As the importance of information grows, the Air Force becomes more reliant on the computer software that processes this information. As this computer software ages and becomes more costly to maintain, the Air Force faces the challenge of finding ways to reduce the cost of software maintenance.

This paper presents several options that can reduce maintenance costs. These options provide a spectrum of software maintenance options that range from little or no change to a software module up to complete replacement of the module. In order to show the effect of these options, the paper first describes how the *understandability* of a software module relates to the difficulty in maintaining the module and thus the cost of maintenance. By examining a software cost estimation model and a reengineering decision model, the factors that affect understandability can be extracted. These factors are presented as one way to quantify the cost of software maintenance. The benefits of using each option in the software maintenance spectrum are described in terms of these factors. Organizations that are faced with maintaining aging software can consider this spectrum of options as alternatives to wholesale replacement of their software.

Background and significance of the problem

As Air Force computer software applications grow older, they generally become harder to maintain. As changes and corrections are made to the software applications, the *legacy code* endures patch after patch. Often the aging application becomes so delicate that even the smallest change can create unpredictable side effects. There are several options short of replacing software modules that may reduce maintenance costs. These options come under the general heading of reengineering and include *redocumentation*, *reverse engineering, translate source code*, and *restructuring*. Redocumentation is a process of generating documentation for the software modules either totally automatically or with some help from the programmer. Reverse engineering is the process of extracting the existing design from a software module in order to help understand what the module is doing. The design is considered to be at a higher level of abstraction than the software itself. Translating the source code can convert a module from an old programming language into a more modern language. Restructuring is used to reorganize the software module making it easier to understand and maintain. A final choice for change is to use the acquisition process to buy a new software application that duplicates the functionality of the aging software application. If done properly, any of these options will reduce the cost of maintenance for a software application.

To appreciate the magnitude of this problem, consider the fact that currently, 30% of the DoD software budget is spent on development, while 70% is spent on maintenance of legacy code.[1] Of these maintenance costs, 50% of the resources are spent on understanding the design and specification of existing systems.[2] Existing and emerging reengineering tools, some of which are fully automated, can help reduce these maintenance costs by providing better documentation of the code through redocumentation, by providing design information through reverse engineering, and by providing a better structure for the code through restructuring. If analysis shows the reengineering effort will not be cost-effective, the final option is to consider replacing the legacy code through new acquisition. All these methods of reengineering or new acquisition are meant to reduce the cost of software maintenance.

Cheyenne Mountain Upgrade (CMU) Example

This section presents one example of how costly the new acquisition of a computer information system can be. The North American Aerospace Defense Command (NORAD) is responsible for providing warning of any missile, air, or space attack. The warning system consists of worldwide missile, atmospheric, and space warning sensors.[3] The hub of this system is located in the Cheyenne Mountain Complex in Colorado Springs, CO. Here, in the Missile Warning Center, the data from all the sensors is processed by computer software and converted into useful information that can be displayed and used for decision making. This system has been operational since 1979 and work began almost immediately to replace and upgrade the computer software.

The Cheyenne Mountain Upgrade (CMU) program was formally started in 1989 under Congressional and Defense Acquisition Board direction.[4] By 1994, the CMU program was eight years behind schedule and $792 million over budget.[5] The software programs that had been delivered were not reliable enough to meet operational needs. As a result, they had to be run simultaneously with the old software programs costing the Air Force an additional $22 million a year. The original replacement systems were scheduled for completion in 1987 at a cost of $968 million.[6] This estimate was moved back to a completion date of 1995 at a total cost of $1.58 billion. The CMU program was finally delivered in 1998 at a cost of $1.7 billion. Table 1 shows more information on the CMU program.

Table 1. Information on Cheyenne Mountain Upgrade

Air Force ACAT IC Program	Cheyenne Mountain Upgrade
Total Number of Systems	One of a Kind
Prime Contractor	Several
Total Program Cost (TY$)	$1770M
Average Unit Cost (TY$)	$1770M
Full-rate production	1QFY94
Rebaselined	4QFY94
Integrated Mission IOT&E	4QFY94
Service Certified Y2K Compliant	No (In progress)

Source: OSD FY98 Annual Report.[7]

The CMU program is an example of how long it can take and how much money can be spent on acquisition of a new software system. With over 2 million source lines of code (SLOC), the CMU program is a very large, complex software system. After almost 10 years and $1.7 billion, the CMU program was finally completed in 1998. The CMU program was intended to replace the software systems previously running in the Cheyenne Mountain Complex. Was new acquisition the only answer to the maintenance problems? Are there other, less time-consuming and less costly options for improving legacy software? After examining factors that drive the cost of software maintenance, this paper presents options across the spectrum of software maintenance that may be more cost-effective than new acquisition.

Notes

[1] Proceedings of the Santa Barbara I Reengineering working group, July 1997, Software Technology Support Center.

[2] Ibid.

[3] GAO AIMD-94-175. Attack Warning: Status of the Cheyenne Mountain Upgrade Program, Letter Report, 09/01/94, US General Accounting Office.

[4] OSD FY98 Annual Report, Office of the Secretary of Defense

[5] GAO AIMD-94-175, Cheyenne Mountain Upgrade

[6] Ibid.

[7] OSD FY98 Annual Report

Part 2

The Importance of Understandability

The computer allows you to make mistakes faster than any other invention, with the possible exception of handguns and tequila.

—Mitch Ratcliffe

In this part of the paper the *understandability* of a software module is presented as a key factor in the cost of software maintenance. If it is hard to understand a software module, it will take longer to fix problems with it, even for a programmer that is quite familiar with the module. This drives up the cost of software maintenance. This effect is presented graphically, showing the significant impact of understandability on a 3000-line software module. Defining and quantifying understandability provides a good start for quantifying the cost of software maintenance.

Software Understanding

The COCOMO II model is a software development tool that can be used to estimate the cost of a new software system. The model estimates the size (in source lines of code or SLOC) and development time (in person-months) required for each software module in the new system.[1] The COCOMO II model includes a formula for estimating the cost of maintaining a newly developed software module.[2] The first part of this formula is based on the idea of *software understanding*, i.e. if the software being maintained is difficult to understand, then more effort will be required to maintain it. The software understanding (SU) factor from the COCOMO II model provides a basis for quantifying the *understandability* of a software module. If the SU factor is high, then the module is easy to understand; if the SU factor is low, then the module is hard to understand. Three factors are considered when determining the value of SU for a software module:

1. **Structure**. Is the code modular and highly cohesive?
2. **Application Clarity**. Is the software a good model of the application?
3. **Self-Descriptiveness**. Is the code well organized and well documented?

The *structure* refers to the modularity and cohesion of a software module. Modularity includes such things as whether the module is one large monolithic program or has been broken into smaller, more reasonably sized modules that each implement part of the solution. Structure is also affected by the number of connections to other modules (coupling), the number of logical functions being performed in the module (cohesion), and the complexity of the software in the module as measured by a complexity metric (reference needed). The *application clarity* aspect refers to how well the software module models the application it implements. The *self-descriptiveness* aspect refers to how well the names given to the variables and sub-programs in the module relate to or match the function being performed. Names that are just single letters of the alphabet are generally not good variable or sub-program names. Self-descriptiveness also refers to the documentation available for the software module.

In COCOMO II, the possible values for the SU factor are:

1. **Very high –** strong modularity, high cohesion, great documentation
2. **High –** low coupling, high cohesion, well commented
3. **Nominal –** reasonable structure, moderately good comments
4. **Low –** high coupling, low cohesion, some useful comments, and documentation
5. **Very low –** spaghetti code, very low cohesion, obscure code and documentation

By examining the structure, application clarity, and self-descriptiveness, a software analyst can determine if the SU factor for a module is very high, low, nominal, etc. This determination is subjective and is based on the opinion of the analyst. There are ways to measure the coupling and cohesion of a module, but the overall quality of the modularity, structure, and documentation is still subjective. There are also cases when the software module does not fall neatly into one of the values presented above. For example, a module may have strong modularity and terrible documentation. In these cases, the analyst must make a subjective decision about this module. The SU factor provides a good start for quantifying the understandability of a software module, but more objective measurements are needed.

Programmer Unfamiliarity

The COCOMO II model also includes *programmer unfamiliarity* in the formula for calculating the cost of maintaining a software module. If the programmers maintaining the software are unfamiliar with a software module, then more effort will be required to maintain it. This unfamiliarity refers simply to the experience a programmer has with a particular module. The effect on the cost of maintenance of the programmer's unfamiliarity with the programming language being used or proper maintenance techniques are considered in Part 3 below. The possible values for the programmer unfamiliarity (UNFM) include *completely familiar*, *mostly familiar*, *somewhat familiar*, *considerably familiar*, *mostly unfamiliar*, and *completely unfamiliar*. These values identify how unfamiliar a programmer is with a particular software module and are used in COCOMO II to estimate the cost of maintaining it.

The Importance of Understandability

This section of the paper compares software understanding and programmer unfamiliarity to show their effect on the cost of software maintenance. By using different values for the software understanding (SU) and programmer unfamiliarity (UNFM) factors to calculate the cost of maintenance, the positive effect understandability has on the cost of maintenance can be shown graphically. The more understandable a software module is, the less it costs to maintain it regardless of how familiar a programmer is with the module.

To show the relationship between SU and UNFM, actual values for these factors were used to calculate different cost of maintenance values. The Costar™ system is a software product that implements the COCOMO II model and was used to determine these cost of maintenance values.[3] In the Costar™ system, the cost of maintenance for a particular software module is expressed in person-months. By fixing the value of SU in a series of data points and changing UNFM from *completely familiar* to *completely unfamiliar*, the effect of understandability on the cost of maintenance can be demonstrated graphically.

The available demonstration version of the Costar™ system limited the size of the software module to 3000 SLOC. The result is that the values calculated for the cost of maintenance are not substantially different from each other, i.e. they may differ by only a few person-months. If a module that is much larger is considered, the subsequent values for the cost of maintenance will also be much larger and the differences between them more substantial[4].

Figure 1 shows the relationship between SU and UNFM. In the figure, the Y-axis is the cost of maintaining the software module as expressed in person-months. The X-axis is the value of the programmer unfamiliarity from *completely familiar* (value 1 on the left) to *completely unfamiliar* (value 6 on the right). The lines in the chart show the different data series where the value of SU was held constant and the value of UNFM was changed. The lowest line is where SU had the value *very high*, the next line up is where SU had the value *high*, and so on. The top line is where SU had the value *very low*.

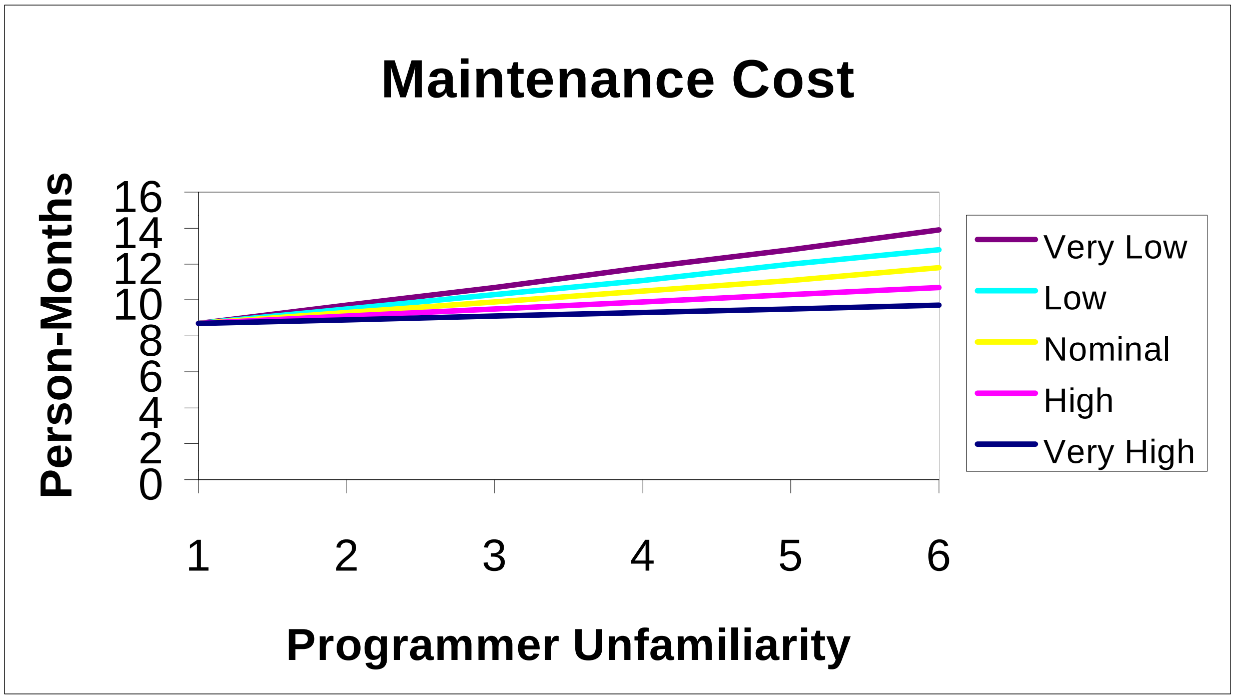

Figure 1. Software Understanding Versus Programmer Unfamiliarity

As shown in the figure, if the value of SU is *very high*, the effect of the level of programmer unfamiliarity for the module is reduced. This means if the understandability for a module is good, i.e. the structure, application clarity, and self-descriptiveness for the module are all very well done, then the cost of maintenance can be reduced. For example, if the SU factor is *very high*, but a maintenance programmer is completely unfamiliar with a software module, the cost of maintenance is still relatively low (see Figure 1). Comparatively, if the SU factor is *very low* and the programmer is completely unfamiliar with the module, then the cost of maintenance is relatively high (see Figure 1).

In this way, the quality of the structure, application clarity, and self-descriptiveness of a software module can reduce the effects of programmer unfamiliarity. If the understandability of a software module is good, then a maintenance programmer with any level of familiarity can maintain the module and the cost of maintenance remains relatively low. The factors that affect structure, application clarity, and self-descriptiveness must be examined in further detail in order to help quantify the cost of software maintenance.

Notes

[1] Barry Boehm and Bradford Clark, Cost Models for Future Software Life Cycle Processes: COCOMO II, USC Center for Software Engineering, Annals of Software Engineering, 1995

[2] Ibid.

[3] Costar™ system, based on COCOMO II model, www.softstarsystems.com

[4] Ibid. The formula for determining the cost of maintenance values was not readily available in the Costar documentation, so it was not possible to tell if the cost grows in a linear or exponential fashion.

Part 3

Quantifying the Cost of Software Maintenance

"No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits…Large system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it…"

—Fred Brooks

This part of the paper describes several factors that can help quantify the cost of maintaining software systems. As seen in the previous section, the *understandability* of a software module is key to the cost of maintaining that module. Factors from the COCOMO II and Software Reengineering Assessment Handbook (SRAH) software cost estimation models that relate to the understandability of a software module are described in more detail here. These factors are collected as a list that can be used to help quantify the cost of maintaining software modules.

The COCOMO II Development Cost Model

The COCOMO II model includes many factors that can have an effect on the *understandability* of a software module. Normally, the COCOMO II model is used to estimate the cost of developing a system of software modules by calculating the length of time required for development and the source lines of code (SLOC) needed for the system. [1] For a more detailed description of the entire COCOMO II model, the reader is referred to Boehm[2]. Since the understandability of a software module is affected by its structure, application clarity, and self-descriptiveness, the factors from COCOMO II that affect them are examined below. These COCOMO II factors are used to help quantify the cost of software maintenance.

For example, the *product complexity* can affect how well the module models the application, which affects application clarity. If the software module being developed is very complex, including many different functions, input screens, and output displays, then, if not designed well, it may be difficult to understand which part of the application is being implemented by which part of the software module. There are programming techniques that can be used to build these complex things in the most efficient manner possible, such as modeling them as *objects* using techniques from the Object-Oriented Paradigm (OOP).[3] But, the degree to which these techniques are used properly and effectively will determine how well a software module models an application. In this way, the complexity of the software module has a direct impact on the application clarity.

The *required reusability* of a software module may improve the structure and self-descriptiveness of the module. If a module is built with the intention of reusing it in future development, then the module will typically be well documented and the inputs, functionality, and outputs of the module will be well defined. This means a module built with reusability in mind will have better structure and better descriptiveness.

The amount of *documentation required* when developing a module clearly affects the self-descriptiveness of the module. A well-documented software module is more understandable because questions about the functionality of the module can be answered by this documentation. Requiring good documentation during the development of a software module is the best way to get accurate, useful documentation.

The *capability of the programmer* has a large effect on the structure, application clarity, and the self-descriptiveness of a software module. If the programmer is not familiar with the best techniques for analysis and design, or does not have good skill at building software modules, then the structure and application clarity of the module will suffer. The programmer's choice for variable and procedure names will have a large impact on the self-descriptiveness as well.

The *development tools* used may impact the structure and application clarity of the software module. If Computer-Aided Software Engineering (CASE) tools are used for the development, such as Rational Apex™, the structure and application clarity of the software module will most likely be good.[4]

The *timeline constraints* can have major impacts on the structure and descriptiveness of the software module. If the development of the software module is under tight time constraints shortcuts in development and documentation may be taken. These shortcuts may degrade the structure of the software module and cause the documentation to be missing or incomplete.

All these factors from the COCOMO II model relate to the understandability of a software module and affect the cost of maintenance.

The SRAH Reengineering Decision Model

The SRAH helps organizations decide when to reengineer legacy software modules. It supports the analysis of organizational, technical, and economic factors in this decision process. Several factors included in the SRAH can affect the understandability of a software module. The SRAH model examines many factors to decide whether or not to reengineer.[5] The factors that are used to decide whether or not a software module has become too hard to maintain can be used to quantify the cost of maintaining that module. These factors can be related directly to the three factors of understandability, i.e. structure, application clarity, and self-descriptiveness.

For example, the *age* of a software module may indicate how modern the structure of the module is. Newer modules were probably built using good, structured programming techniques. Older modules may have been built with poor structures and then maintained by several different programmers over time. If the software has been developed within the last five years, then chances are good that it was developed using a modern language and that the development staff is still available to answer maintenance questions.[6] As software modules age, they may be changed and fixed by several different programmers. As this continues, the structure of the module tends to suffer and become harder to understand.

Complexity measurement tools measure the *complexity* of a software module by looking at the structure of a software module. Modules with many decision points and branches are considered complex, so the complexity of a software module is directly related to its structure. Modules with complex structures are harder to understand because of the difficulty in following the decision points and branches. As the complexity of a software module increases it becomes harder to understand.

The *language* factor relates directly to the structure of a module as well as the application clarity. Older, lower-level languages such as FORTRAN do not provide many tools for expressing the solution to a problem in abstract, general ways.[7] This means the overall structure of the module will be based on low-level, detailed constructs and it may be hard to understand which part of the application this module implements. Newer, higher-level languages such as Ada, Java, and C++ provide the programmer with many more tools for abstraction.[8] If the application being modeled is an airplane, for example, Ada allows the programmer to associate the data and behavior of the parts of the airplane together in one abstract unit. This improves the structure of modules if done properly, because the solution to a problem can be split into logical pieces that correspond to the application being modeled. Using a newer, higher-level language improves the structure and application clarity of software modules.

As discussed in the previous section, the *documentation* provided for a module affects its understandability. If the documentation is missing or completely inaccurate, then the programmer must rely on his/her own ability to read the code and determine the functionality of the software module. The documentation for a software module is a big part of its self-descriptiveness and its understandability.

The *personnel knowledge* affects the structure and the descriptiveness of the software module. If the programmer maintaining the software module is not an experienced maintenance programmer, he/she may degrade the structure of the module or unnecessarily add to its complexity. If different programmers maintain a single module over time, their differing personal techniques will tend to slowly degrade its structure as well. If the documentation is not changed along with the software module, it will slowly become inaccurate and of little use.

These factors from the SRAH model affect the structure, application clarity, or descriptiveness of a software module, thus affecting the understandability of the module.

The Cost of Software Maintenance

The factors from the COCOMO II and SRAH models that affect the understandability of a software module provide a good way to quantify the cost of software maintenance. The understandability of a software module has a large effect on the cost of its maintenance (as shown in Part 2 above). The COCOMO II and SRAH factors affect understandability because of their effect on the structure, application clarity, and self-descriptiveness of a software module. This part of the paper lists these factors and describes how they quantify the cost of software maintenance.

Table 2 – COCOMO II and SRAH Factors

COCOMO II	SRAH
Product complexity	Complexity
Required reusability	
Documentation required	Documentation
Capability of the programmer	Personnel knowledge
Development tools used	Language
Timeline constraints	
	Age

The table above shows the factors from the COCOMO II and SRAH models that affect understandability. The factors have been arranged in the table in order to highlight similarities. For example, the SRAH factor *personnel knowledge* is similar to the COCOMO II factor of *capability of the programmer*. Even though the COCOMO II model is intended to be used in software development and the SRAH model is intended for maintenance and reengineering, they both include similar factors that affect the understandability of a software module. These factors from both models can be combined to quantify the cost of software maintenance.

The following table lists the factors that affect and quantify the cost of software maintenance. In the table, the *complexity* factor refers to the sophistication of the module in terms of menus, input/output, and functionality, as well as, the complexity of the structure as measured by complexity measurement tools. The *reusability* factor refers to how easily a software module could be adapted for reuse in another application. The *documentation* factor refers to the quality of the original documentation effort and the quality of the existing documentation. The *capability of the programmer* factor refers to the expertise of the original designer as well as the programmers that maintain the software module. The *development tool and language use* factor refers to the prudent use of CASE tools, programming environments, and modern languages in development and maintenance. The *timeline constraints* factor refers to the effect of working under short timeline pressures. The *age* factor refers to how long the module has been maintained.

Table 3 – Factors in the Cost of Software Maintenance

Factor	Decrease	Increase
Complexity	Low	High
Reusability	High	Low
Documentation	High	Low
Capability of the programmer	High	Low
Development tool and Language use	Good	Poor
Timeline constraints	Few	Many
Age	Young	Old

For all these factors, the values in the *decrease* column indicate how this factor will help decrease the cost of maintenance. For example, low *complexity* helps decrease the cost of maintenance. Similarly, the values in the *increase* column indicate how the factor will drive up the cost of maintenance. For example, low *reusability* would increase the cost of maintaining the module.

The factors listed in Table 3 affect the understandability of software modules and quantify the cost of software maintenance. These factors were built by examining factors from both the COCOMO II software development model and the SRAH reengineering decision model. Since the factors in Table 3 affect the structure, application clarity, or self-descriptiveness of a software module, they can increase or decrease the cost of maintenance. These factors will be used in the following section to quantify the benefits of each option across the software maintenance spectrum.

Notes

[1] Barry Boehm and Bradford Clark, Cost Models for Future Software Life Cycle Processes: COCOMO II, USC Center for Software Engineering, Annals of Software Engineering, 1995.

[2] Ibid.

[3] Grady Booch. Object-Oriented Analysis and Design (2nd Edition). Redwood City, CA: The Benjamin/Cummings Publishing Co, 1994.

[4] *Rational Apex*, product of Rational Software Corporation, www.rational.com

[5]. The Software Reengineering Assessment Handbook (SRAH). JLC-HDBK-SRAH. March 1997. In conjunction with the Software Technology Support Center (STSC)

[6] Ibid.

[7] REFINE/FORTRAN user's guide, Reasoning Systems Inc, Palo Alto, CA. 1994.

[8] J.G.P. Barnes, *Programming in Ada*, Wokingham, England: Addison-Wesley, 1994. Deitel & Deitel, *How to Program Java*, Prentice Hall, Upper Saddle River, NJ, 1998. Bjarne Stroustrup, *The C++ Programming Language*, ATT Bell Labs, New Jersey, Jul 1987.

Part 4

The Software Maintenance Spectrum

Computer /n./: a device designed to speed and automate errors.

—Jargon File

This part of the paper presents a spectrum of options for maintaining legacy software modules. The benefits of these options are quantified in terms of how they can decrease the cost of maintenance. Each option's effect on the factors in the cost of maintenance (presented in Table 3) will be discussed below. These options provide a full spectrum of choices for organizations maintaining software modules.

The SRAH report lists six options for reengineering: *redocument*, *reverse engineer*, *translate source code*, *data reengineer*, *restructure*, and *retarget*.[1] The *data reengineer* option, which refers to reengineering database products, and the *retarget* option, which is a process used to move a software module from one hardware platform to another, were not included in this study. Thus, the options considered in this study were limited to *redocument*, *reverse engineer*, *translate source code*, and *restructure*.

The *restructure* option can be further broken down based on whether or not the restructuring will shift the software module from one *paradigm* to another. A programming paradigm refers to the way in which a software module is designed and implemented. Structured analysis and design, typically based on functionality, is used in the Imperative paradigm with such languages as Pascal, C, and FORTRAN.[2] Object-oriented analysis and design is used in the Object-

Oriented Paradigm with such languages as Ada, Java, and C++.[3] Recently, restructuring tools have been developed to convert modules from the Imperative paradigm to the Object-Oriented Paradigm.[4] Thus, the restructure option is broken into *restructure within a paradigm* and *restructure into a new paradigm*.

The SRAH report also lists two classic maintenance strategies: *redevelopment* and *status quo*.[5] The redevelopment option is called *new acquisition* in this study in order to tie this option directly to the Air Force acquisition process. The status quo option means to continue maintaining the software module without reengineering or doing new acquisition.

The spectrum of software maintenance considered in this study includes *status quo, redocument*, *reverse engineer*, *translate source code*, *restructure within a paradigm*, *restructure into a new paradigm*, and *new acquisition*. These options are listed from lowest to highest in order of the amount of change (and consequent cost) the existing software module requires.

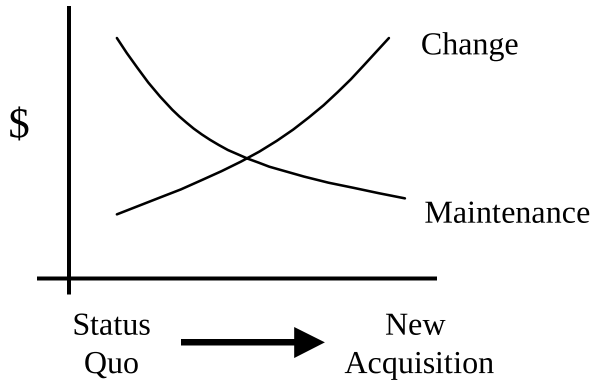

Figure 2 - Cost of Change versus Cost of Maintenance

In relative terms, Figure 2 shows the effect on the amount and cost of change versus the effect on the cost of maintenance as the options move across the spectrum from *status quo* to *new acquisition*. As more time and money is invested, the cost of maintenance is expected to decline. The benefits of these options are discussed below in terms of their effect on understandability (Figure 1) and the factors that affect the cost of maintenance (Table 3).

Status Quo

The *status quo* option, which means no reengineering or new acquisition will be done, is a reasonable choice for the maintenance of software modules with little remaining life. The SRAH model states that if a software module is expected to be used for three years or less, the cost of reengineering the module may not be recovered in the life of the module and the organization should consider not reengineering it.[6]

Redocument

The *redocument* option involves generating new documentation for a software module. Some tools are available to help automate this process.[7] The benefit of redocumenting is that it improves the *documentation* factor (see Table 3) and improves a software module's self-descriptiveness. This improves the understandability of a software module. For example, if the understandability of a software module is *low*, redocumenting may make the understandability *nominal* or *high*. As shown in Figure 1, the effects of programmer unfamiliarity are reduced when understandability is *high* and the cost of maintenance will be reduced. The decision to redocument is subjective and based on analysis of the current documentation. If the current documentation is poor and redocumentation is done properly, the redocument option may be an effective way to reduce the cost of maintaining this module.

Reverse Engineer

The *reverse engineer* option involves extracting design-level information from the software module and may be done automatically or semi-automatically.[8] Design-level information may include the current organization of and communication between modules in the system. It may also include descriptions of the expected input and output of a module. The design-level information can be added to the module's documentation and may improve the *documentation* factor (see Table 3) of the module. This can improve the application clarity or self-descriptiveness of the module, thus improving the understandability. As shown in Figure 1, improving the understandability will reduce the cost of maintenance. The amount of improvement gained from reverse engineering is subjective and differs for each module. If the application clarity and self-descriptiveness can be improved, the reverse engineering option may be a cost-effective means of reducing the cost of maintenance.

Translate Source Code

The *translate source code* option, which involves converting the software module from one programming language to another, may be done automatically and is a low-level transformation.[9] Translating the source code is not considered a significant or beneficial improvement because it is usually obvious that the finished product has been translated from another language.[10] Unless careful steps are taken to take advantage of the new language, the improvement will be minimal. However, translating the source code may positively affect the *capability of the programmer* factor (see Table 3) if the programmer is familiar with the new language. It may also affect the *development tool and language use* factor (see Table 3) if there are tools available for the new language. The translate source code option is not the most desirable option, but it may have positive affects on factors that will reduce the cost of maintenance.

Restructure within a Paradigm

The *restructure within a paradigm* option involves changing the structure of a module or set of modules while keeping them in the same programming language paradigm. For example, a system that includes modules written in the FORTRAN programming language could be restructured to improve the structure of the modules while keeping them in FORTRAN language. These improvements could include such things as improving the coupling and cohesion of the modules or removing code that is not being used. These changes may improve the complexity of the modules, and their reusability, thus improving their understandability. As shown in Figure 1, this will reduce the cost of maintenance. The positive effects restructuring has on the understandability of software modules make it a viable option for reducing the cost of software maintenance.

Restructure into a New Paradigm

The *restructure into a new paradigm* option involves restructuring a module or collection of modules and moving them from one programming language paradigm to another. This process is challenging because the fundamental ways of solving the problem change from one paradigm to another. Recently, the Object-Oriented Paradigm (OOP) with the Ada, Java, and C++ languages has become more widely used. This part of the paper will briefly examine the OOP and quantify the benefits of using the OOP in terms of the factors that affect the cost of maintenance.

The OOP is based on the ideas of objects, classes, methods, and inheritance.[11] An *object* models the data and behavior of something from the application domain. For example, when building a flight simulator, airplanes, instruments, and airports are objects from the application domain. Objects are built from *classes* which define the data, i.e. attributes for the objects. For example, the number of engines is an attribute of an airplane. *Methods* are the pieces of programming language code which implement the behavior of the object such as "bank left" for an airplane. *Inheritance* is an abstraction mechanism that allows common attributes and behavior to be shared between classes. This brief description of the OOP is meant to provide a common framework for discussion. For further information on the OOP, the reader is referred to Booch.[12]

One of the benefits of the OOP includes its inherent ability to model the real world application using the terminology from the application domain.[13] Classes and objects get their names from the application domain, which builds a direct link from the application domain to the software being developed. This link improves the *documentation* factor (see Table 3) by inherently providing documentation of what object the software module is modeling. The inherent documentation has a positive effect on the application clarity and improves the understandability of the software module.

For similar reasons, the self-descriptiveness of a software module can be improved because of the link between the application domain and the classes and objects being developed. By naming the classes and objects in the OOP software after objects in the application domain, the classes and objects can be more easily understood. This naming convention helps a maintenance programmer understand the intended functionality of these parts of the software system. The names help document the functionality of the software increasing the self-descriptiveness and understandability of the module.

Using the *restructure into a new paradigm* option may increase the reusability of software modules. If done properly, the interface to the classes and objects that implement these modules will become clearer. This may increase the chances that another programmer will understand the functionality and can reuse the classes and objects for another application. For example, an airplane class could be adapted from a flight simulator and reused in an air traffic control system. Improving the *reusability* factor (see Table 3) improves the understandability of the software module and can help reduce the cost of maintaining the module.

The *restructure into a new paradigm* option may also have a positive effect on understandability base on the expertise of the maintenance programmers. For example, if an organization has programmers that are predominantly skilled at maintaining OOP software, then converting from the Imperative paradigm to the OOP will allow these programmers to use their skills more effectively. This improves the *capability of the programmer* factor (see Table 3) and may have a positive effect on the understandability of the software modules and reduce the cost of maintenance.

The *restructure into a new paradigm* option may also improve the understandability of the software modules because of the tools available for maintaining OOP software. For example, an organization can use tools to edit and maintain the attributes and methods of classes and the overall structure of the new OOP system. This will increase the *development tool and language use* factor (see Table 3) and improve the understandability of the modules to reduce the cost of maintenance.

The benefits of restructuring software modules into the OOP include improvements in the application clarity, self-descriptiveness, inherent documentation, reusability, capability of the programmer, and tools available for maintenance. All these factors combine to improve the understandability of the software modules. As presented in Figure 1, improving the understandability helps reduce the cost of software maintenance.

Because of the many different factors that restructuring to the OOP can affect, there may be large improvements in the understandability of the software modules. For example, if properly done restructuring to the OOP can change the understandability from *very low* to *high*. Some systems are available for automatically or semi-automatically converting from the Imperative paradigm to the OOP. If these systems are used, the restructure into a new paradigm option can be a cost-effective way to reduce the cost of software maintenance.

New Acquisition

If a software module is replaced through *new acquisition*, then the cost of maintenance for this new module should be reduced. If done properly, all the factors from Table 3 can be improved and the module can have very good understandability. The new acquisition option has the most positive effect on the cost of maintenance, but it also makes the most changes and may take the most time and money (see Figure 2). The Cheyenne Mountain Upgrade example (see Table 1) demonstrates how new acquisition projects may take several years and cost millions of dollars. An organization should carefully consider the benefit versus cost of using the new acquisition option.

Challenges

Some of the challenges with using the entire software maintenance spectrum include technology transfer, quality, and scalability. Some of the reengineering techniques are only academic products and are not ready for commercial use. There must be some investment in the transfer of these ideas and technology to industry before they can be used. If the quality of the reengineered product is poor, the cost of maintenance may not be reduced. Some of the reengineering techniques are not able to handle large software systems. For some systems, it is impossible to operate on a system the size of the Cheyenne Mountain Upgrade with its 2 million lines of code. These challenges continue to be addressed as the reengineering technologies mature.

Notes

[1] The Software Reengineering Assessment Handbook (SRAH). JLC-HDBK-SRAH. March 1997. In conjunction with the Software Technology Support Center (STSC)

[2] Herbert L. Dershem and Michael J. Jipping, Programming Languages: Structures and Models. Boston, MA: PWS Publishing Co, 1993.

[3] Grady Booch. Object-Oriented Analysis and Design (2nd Edition). Redwood City, CA: The Benjamin/Cummings Publishing Co, 1994.

[4] Ivar Jacobson and Fredrik Lindstrom. *Reengineering Old Systems to an Object-Oriented Architecture*. OOPSLA Proceedings, 1991. Phillip Newcomb,. *Reengineering Procedural into Object-Oriented Systems*. In 2nd Working Conference on Reverse Engineering, Los Alamitos, CA. IEEE Computer Society Press, Jul 1995. Ricky E. Sward, *Extracting Functionally Equivalent Object-Oriented Designs from Legacy Imperative Code*. PhD Dissertation. Air Force Institute of Technology. 1997.

[5] SRAH

[6] Ibid.

[7] *+1Reports* program, +1 Software Engineering, Carmarillo, CA, www.plus-one.com

[8] Cradle system, Craven House, Cumbria, LA, www.threesl.com

[9] Robert Vienneau, A Review of Non-Ada to Ada Conversions, Report to Rome Laboratory, RL/C3C, Rome, NY, August 1993.

[10] Ricky E. Sward, PhD Dissertation.

[11] Grady Booch. Object-Oriented Analysis and Design

[12] Ibid.

[13] Ibid.

Part 5

Conclusions

I have traveled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year.

—Prentice-Hall Editor, 1957

As the Air Force enters the 21st century, information processing is here to stay. The Air Force continues to pursue information superiority in the battlespace. As our computer software follows us kicking and screaming into the 21st century, we will continue to face the task of maintaining legacy software. Air Force organizations should consider the entire spectrum of software maintenance options when maintaining legacy software.

This paper used the COCOMO II software development cost estimation model and the SRAH reengineering decision model to demonstrate the importance of *understandability*. Figure 1 showed how understandability can affect the cost of software maintenance. If a software module is easy to understand, the effects of unfamiliar programmers maintaining the module can be minimized. The structure, application clarity, and self-descriptiveness of a software module contribute directly to its understandability. Several factors from the COCOMO II and SRAH models[1] were culled out and presented as factors that affect these three things. These factors were combined and presented in Table 3 as those factors that quantify the cost of software maintenance.

The full spectrum of software maintenance options was presented and benefits of each option were quantified in terms of factors from Table 3. These options include *status quo*, *redocument*, *reverse engineer*, *translate source code*, *restructure in the same paradigm*, *restructure into a new paradigm*, and *new acquisition*. Figure 2 showed how as more time and money is spent along this spectrum of options, the cost of maintenance is expected to decline. Each option has unique effects on the factors that drive the cost of software maintenance and the benefit of each option was presented in terms of these factors.

Organizations that are faced with the challenge of maintaining legacy software should consider the full spectrum of software maintenance options. Reengineering techniques offer the ability to redocument poorly documented code, reverse engineer the design of the code, translate the source code to a more maintainable language, restructure the code in the same paradigm, restructure the code into a new paradigm, or buy new software. If automated systems are used to accomplish these tasks, in most cases they will provide more cost-effective ways to reduce the cost of software maintenance without requiring total replacement of the module. In this way, organizations can avoid the cost of time and money spent on new acquisition. Organizations need to consider the effect these options will have on their software modules in order to reduce the cost of software maintenance (see Table 3).

The Air Force faces many challenges in the 21st century and maintaining legacy software in an age of ever-increasing dependence on information is one of them. Organizations faced with the challenge of maintaining legacy software should consider the full spectrum of reengineering options before pursuing costly new acquisitions.

Notes

[1] The Software Reengineering Assessment Handbook (SRAH). JLC-HDBK-SRAH. March 1997. In conjunction with the Software Technology Support Center (STSC) Barry Boehm and Bradford Clark, *Cost Models for Future Software Life Cycle Processes: COCOMO II*, USC Center for Software Engineering, Annals of Software Engineering, 1995.

Glossary

Cohesion. The extent to which a software module is focused on performing a single function. A module with good cohesion will focus on one or two tightly related functions. A module with poor cohesion will perform many unrelated functions.

Coupling. The extent to which a software module has connections to other software modules. This can be measured by counting the number of calls the software module makes to other modules.

Data reengineering. Tools that perform all the reengineering functions associated with source code (reverse engineering, forward engineering, translation, redocumentation, restructuring/normalization, and retargeting) but act upon data files.

Domain objects. Things in the problem domain that take on identity and behavior. For example, an aircraft has a unique identity, e.g. a tail number, and behavior, e.g. level flight.

Forward Engineering. Forward engineering is the set of engineering activities that consume the products and artifacts derived from legacy software and new requirements to produce a new target system.

Legacy code. Aging software applications that are hard to maintain. They are often complex, unstructured, and include no documentation.

Object-oriented programming. The programming language paradigm that designs software applications to model domain objects with data and behavior.

Redocumentation. The process of analyzing the system to produce support documentation in various forms including users manuals and reformatting the systems' source code listings. A special case of redocumentation tools are reformatting tools. Otherwise known as "pretty printers", reformatters make source code indentation, bolding, capitalization, etc. consistent thus making the source code more readable.

Reengineering The examination and alteration of an existing subject system to reconstitute it in a new form. This process encompasses a combination of sub-processes such as reverse engineering, restructuring, redocumentation, forward engineering, and retargeting.

Restructuring. The engineering process of transforming the system from one representation form to another at the same relative abstraction level, while preserving the subject system's external functional behavior.

Retargeting. The engineering process of transforming and hosting or porting the existing system in a new configuration.

Reverse Engineering. The engineering process of understanding, analyzing, and abstracting the system to a new form at a higher abstraction level.

Source Code Translation. Transformation of source code from one language to another or from one version of a language to another version of the same language (e.g., going from COBOL-74 to COBOL-85).

Bibliography

Bergey, John, Dennis Smith, and Nelson Weiderman. DoD Legacy System Migration Guidelines. Software Engineering Institute Technical Report CMU/SEI-99-TN-013.

Bergey, John, Dennis Smith, Nelson Weiderman, and Steven Woods. Options Analysis for Reengineering (OAR): Issues and Conceptual Approach. Software Engineering Institute Technical Report CMU/SEI-99-TN-014.

Barnes, J.G.P. *Programming in Ada*, Wokingham, England: Addison-Wesley, 1994.

Barry Boehm and Bradford Clark. *Cost Models for Future Software Life Cycle Processes: COCOMO II*, USC Center for Software Engineering, Annals of Software Engineering, 1995

Booch, Grady. *Object-Oriented Analysis and Design* (2nd Edition). Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1994.

Byrne, Eric J. *A conceptual foundation for software reengineering*. In Proceedings of the International Conference on Software Maintenance. IEEE Computer Society Press. pp. 216-235, Nov 1992.

Chikofsky, Elliot and James Cross. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, 7(1):13-17 (Jan 1990).

Deitel & Deitel, *How to Program Java*, Prentice Hall, Upper Saddle River, NJ, 1998.

Dershem, Herbert L and Michael J. Jipping, Programming Languages: Structures and Models. Boston, MA: PWS Publishing Co, 1993.

GAO AIMD-94-175. *Attack Warning: Status of the Cheyenne Mountain Upgrade Program*, Letter Report, 09/01/94, US General Accounting Office

Jacobson, Ivar and Fredrik Lindstrom. *Reengineering Old Systems to an Object-Oriented Architecture*. OOPSLA Proceedings. pp. 340-350. 1991.

Korson, Tim and John D. McGregor. *Object-Oriented: A Unifying Paradigm*. Communications of the ACM. 33(9):40-60. Sep 1990.

Newcomb, Phillip. *Reengineering Procedural into Object-Oriented Systems*. In 2nd Working Conference on Reverse Engineering, Los Alamitos, CA. IEEE Computer Society Press, Jul 1995, 237-251.

Olsem, Michael R. and Chris Sittenauer. *Reengineering Technology Report*. Technical Report Vol 1, Hill AFB, UT: Software Technology Support Center, Aug 1993. 17 Jul 97.

OSD FY98 Annual Report. Annual Report for FY98. Office of the Secretary of Defense.

Quilici, Alex. *Reverse Engineering of Legacy Systems: A Path Toward Success*. In Proceedings of the 17th Annual International Conference on Software Engineering. IEEE Press. Seattle, WA, pp. 331-336, April 1995 (invited position paper).

Sneed, H. *Migration of Procedurally Oriented COBOL Programs in an Object-Oriented Architecture*. Proceedings of the Conference on Software Maintenance. pp. 105-116. Nov 1992.

Sneed, Harry M. and Erika Nyary. *Extracting Object-Oriented Specifications from Procedurally Oriented Programs*. In 2nd Working Conference on Reverse Engineering, Los Alamitos, CA. IEEE Computer Society Press, Jul 1995, pp. 217-226.

Sneed, Harry M. and Agnes Kaposi. A Study on the Effect of Reengineering on Maintainability. International Conference on Software Maintenance, 1990, IEEE Society, pp 91-99.

SRAH. The Software Reengineering Assessment Handbook. JLC-HDBK-SRAH. March 1997. In conjunction with the Software Technology Support Center (STSC)

Stroustrup, Bjarne. *The C++ Programming Language*, ATT Bell Labs, New Jersey, Jul 1987.

Sward, Ricky E. *Extracting Functionally Equivalent Object-Oriented Designs from Legacy Imperative Code*. PhD Dissertation. Air Force Institute of Technology. 1997.

Vienneau Robert, *A Review of Non-Ada to Ada Conversions, Report to Rome Laboratory*, RL/C3C, Rome, NY, August 1993.

DISTRIBUTION A:

Approved for public release; distribution is unlimited.

Air Command and Staff College
Maxwell AFB, Al 36112