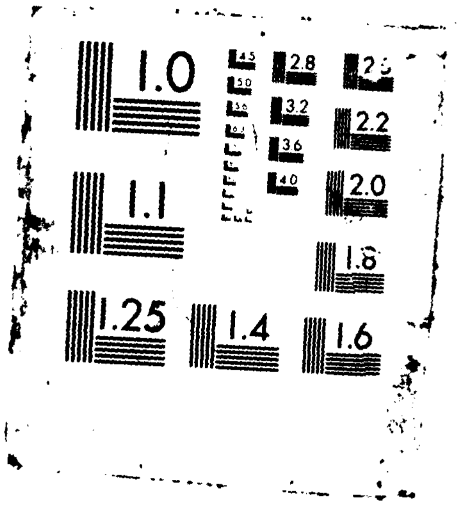
AD-A170 528 SOFTWARE MODERNIZATION(U) AIR WAR COLL MAXWELL AFB AL 1/1
R C HANLON MAY 86 AU-AWC-86-088

UNCLASSIFIED F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(10)

AIR WAR COLLEGE

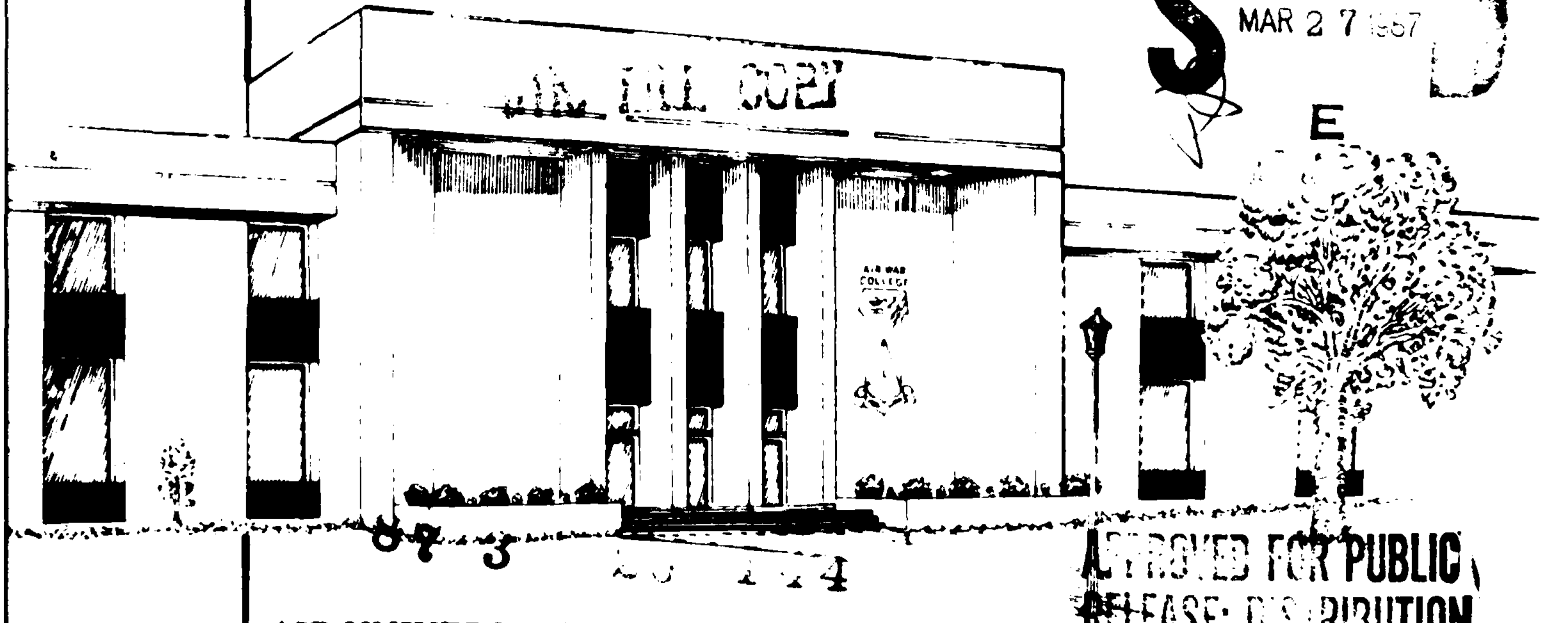
RESEARCH REPORT

No. AU-AWC-86-088

AD-A178 528

SOFTWARE MODERNIZATION

By LT COL ROBERT C. HANLON



87 3 26 234

AIR UNIVERSITY
UNITED STATES AIR FORCE
MAXWELL AIR FORCE BASE, ALABAMA

AIR WAR COLLEGE
AIR UNIVERSITY

SOFTWARE MODERNIZATION

by

Robert C. Hanlon
Lieutenant Colonel, USAF

A RESEARCH REPORT SUBMITTED TO THE FACULTY

IN

FULFILLMENT OF THE RESEARCH

REQUIREMENT

Research Advisor: Colonel Richard D. Clark

MAXWELL AIR FORCE BASE, ALABAMA

May 1986

DISCLAIMER-ABSTAINER

This research report represents the views of the author and does not necessarily reflect the official opinion of the Air War College or the Department of the Air Force.

This document is the property of the United States government and is not to be reproduced in whole or in part without permission of the Commandant, Air War College, Maxwell Air Force Base, Alabama.

AIR WAR COLLEGE RESEARCH REPORT ABSTRACT

TITLE: Software Modernization

AUTHOR: Robert C. Hanlon, Lieutenant Colonel, USAF

The dependence of the United States on qualitative superiority to maintain deterrence has increased emphasis on software for flexible and responsive support of mission critical and general-purpose applications. However, responsiveness and integrity of software systems is being jeopardized by the labor-intensiveness of software programming and growing shortage of qualified software programmers relative to expanding demand.

Although the software problem is generally recognized and initiatives have been started to improve the situation, progress has failed to reverse the trend. Technological solutions offer potential to reverse this situation. However, near-term gains are limited by attitudes and approaches toward software modernization. Changes are needed to minimize near-term problems and accelerate achievement of long term benefits. The most critical near term action is the establishment of progressive and demanding policies on software modernization.

Factors which make software programming labor-intensive and which impact on software modernization are presented along with some ongoing efforts to reduce this labor-intensiveness and provide for software modernization. Recommendations on areas requiring additional emphasis to address this critical problem are suggested.

BIOGRAPHICAL SKETCH

Lieutenant Colonel Robert C. Hanlon received a B.S. in engineering from Case-Western Reserve University, Cleveland, Ohio. After graduation in 1968, he attended Officer Training School, Lackland AFB, Texas, and was commissioned in 1969. After attending the communications-electronics officer course at Keesler AFB, Mississippi, he was an operations staff officer at HQ Air Force Communications Service, Scott AFB, Illinois, and Richards-Gebaur AFB, Missouri. In 1973, while at Richards-Gebaur AFB, Colonel Hanlon earned an M.S. in electrical engineering from the University of Kansas. After serving as the Chief of Operations, 2129 Communications Squadron, Ching Chuan Kang AB, Taiwan, he was assigned as Chief of Logistics Inspection, HQ Tactical Communications Area, Langley AFB, Virginia. At Langley AFB, he also served as Chief of Maintenance, 1913 Communications Squadron. He then had an AFIT assignment to the University of Illinois to obtain a Ph.D. in electrical engineering. In 1980, Colonel Hanlon went to the Command and Control Technical Center, Defense Communications Agency, and also served in the HQ and the National Communications System. Prior to attendance at Air War College, he was with HQ USAF, Directorate of Command, Control, and Telecommunications, and the Assistant Chief of Staff for Information Systems. Colonel Hanlon is married to the former Sharon Ann Evans of Euclid, Ohio. They have one daughter, Kelly Elizabeth. Lieutenant Colonel Hanlon is a graduate of the Air War College, class of 1986.

TABLE OF CONTENTS

CHAPTER		PAGE
	DISCLAIMER-ABSTAINER	ii
	ABSTRACT	iii
	BIOGRAPHICAL SKETCH	iv
	LIST OF ILLUSTRATIONS	vii
	LIST OF TABLES	viii
I	INTRODUCTION	1
II	THE MODERNIZATION PROBLEM	6
	Standards	6
	Portability	8
	Automated Support Tools	8
	Documentation	10
	Software Architectures	12
	Non Technical Factors	13
III	ONGOING APPROACHES	14
	Software Technology for Adaptable, Reliable Systems	14
	Ada Programming Language	15
	End User Computing	16
	Data Base Management Systems	18
IV	RECOMMENDATIONS	21
	People Efficiency Over Machine Efficiency	21
	Fewer and Better Trained People	24
	Interoperability and Networking	26
	Policy Approach	27

V CONCLUSION 29

LIST OF REFERENCES 30

GLOSSARY 32

LIST OF ILLUSTRATIONS

Figure 1 Trends in Software Supply and Demand 5

LIST OF TABLES

Table 1 Automated Support Tools 10

CHAPTER I
INTRODUCTION

The military strategy of the United States (US), particularly the element of deterrence, depends heavily on maintaining qualitative superiority over its potential adversaries. Qualitative superiority compensates for real or perceived quantitative shortcomings in US forces. This dependence on qualitative superiority is reflected in heavy use of advanced technology in most--if not all--major US weapon systems, and is seen in extensive use of programmable digital computers.

"Computers embedded in mission critical military systems are integral to [US] strategic and tactical capabilities.... The military power of the United States is inextricably tied to the programmable digital computer." (1:viii)

The functioning of a programmable digital computer is defined and controlled by its software. "Software is the embodiment of system 'intelligence'." (1:viii) More specifically, software is that element of the overall system consisting of three *necessary* components: (2:31)

• computer program code – the instructions to the machine, both as written (i.e., source code), and as translated and used by the machine (i.e., object code);

• data – information stored in the machine and used to provide an input to augment, control, or modify the instructions to the machine; and

• documentation – information used throughout the life cycle (development, operation, and maintenance) of the software subsystem to understand and control the form, substance, and use of the software.

The critical importance of software to a system extends beyond its functional role. The cost, reliability, and time required to field new systems and applications are increasingly determined by software rather than hardware considerations. The cost of software to the Department of Defense (DoD) is estimated to be four to eight billion dollars ($4,000,000,000– $8,000,000,000) per year. (3:78) Of importance beyond this large aggregate sum, software costs of a major system can exceed 80 percent of the total system cost. This percentage has risen systematically from 20 percent and continues to grow. (2:29) As a major--and even dominant--element of a system, the software is frequently on the critical path for system development, and has often been late, or over budget, or both. (1:ix)

This cost and criticality of software in US weapon systems raises the question: why is this reliance on software tolerated--much less allowed to expand? The fundamental reason is to support change. Software is "soft" because it can be "easily" changed. Consequently, the system can be "easily" changed. Further, despite the magnitude of the cost figures cited above, software changes are less costly to make than the corresponding physical system changes. The United States Air Force (USAF) experience with the F-111 aircraft program highlights this:

> Similar avionics capabilities were implemented in analog electronic hardware on the F-111 A/E and in software on the F-111 D/F. A number of changes were tracked through both systems. The savings in dollars and deployment lead-time in the digital F-111 D/F are striking. *Hardware changes cost fifty times as much as software changes and took three times as long to make.* [emphasis added] (1:3)

Although the extent of the role of software in systems' functionality continues to expand, the greatest portion of effort and cost spent on software occurs after its development, during the software maintenance phase. Software maintenance can account for 70 percent of the total system costs and 40 to 95 percent of the overall personnel effort. (2:3) This would clearly be grossly excessive if software maintenance consisted solely of the correction of errors that went undetected during its development and testing. In fact, only about 18 percent of the software maintenance effort is directed toward correcting errors. The majority (82 percent) of the software maintenance effort encompasses changes to the software resulting from altered or additional requirements specifications. (2:30)

This high proportion of software maintenance costs reflects the fact that change is an inherent characteristic of all software. Such changes are required to improve performance, extend functionality, adapt to processing environment changes, and accommodate changing user needs. To the extent that such changes are justifiable on a cost-versus-benefit basis to meet minimum-essential, validated requirements, there is no basic problem with such a high percentage of system costs being expended for software maintenance. However, there are insufficient qualified software programming specialists available to support these burgeoning requirements. This can be seen dramatically in Figure 1 which shows the growing shortages of qualified personnel. Consequently, demands for support of existing systems impact upon the ability to develop new systems, to include modernization of existing software. "And unless radical new methods are found, maintenance will go even higher in its demands and will very nearly stifle further development." (2:29)

Although the software problem is generally recognized and initiatives have been started to improve the situation, progress has failed to reverse the trend. Technological approaches offer significant potential to reverse this situation. However, near-term gains are limited by existing attitudes and approaches toward software modernization. Fundamental changes are needed to minimize near-term problems and accelerate the achievement of long-term benefits. The most critical near term step in reversing the negative trends is the establishment of progressive and demanding policies on software modernization.

Chapter II outlines some of the factors which make software programming so labor-intensive and which impact on software modernization. Chapter III then looks at some of the ongoing efforts to reduce this labor-intensiveness and provide for software modernization. Chapter IV offers some recommendations on areas requiring additional emphasis to address this critical problem.

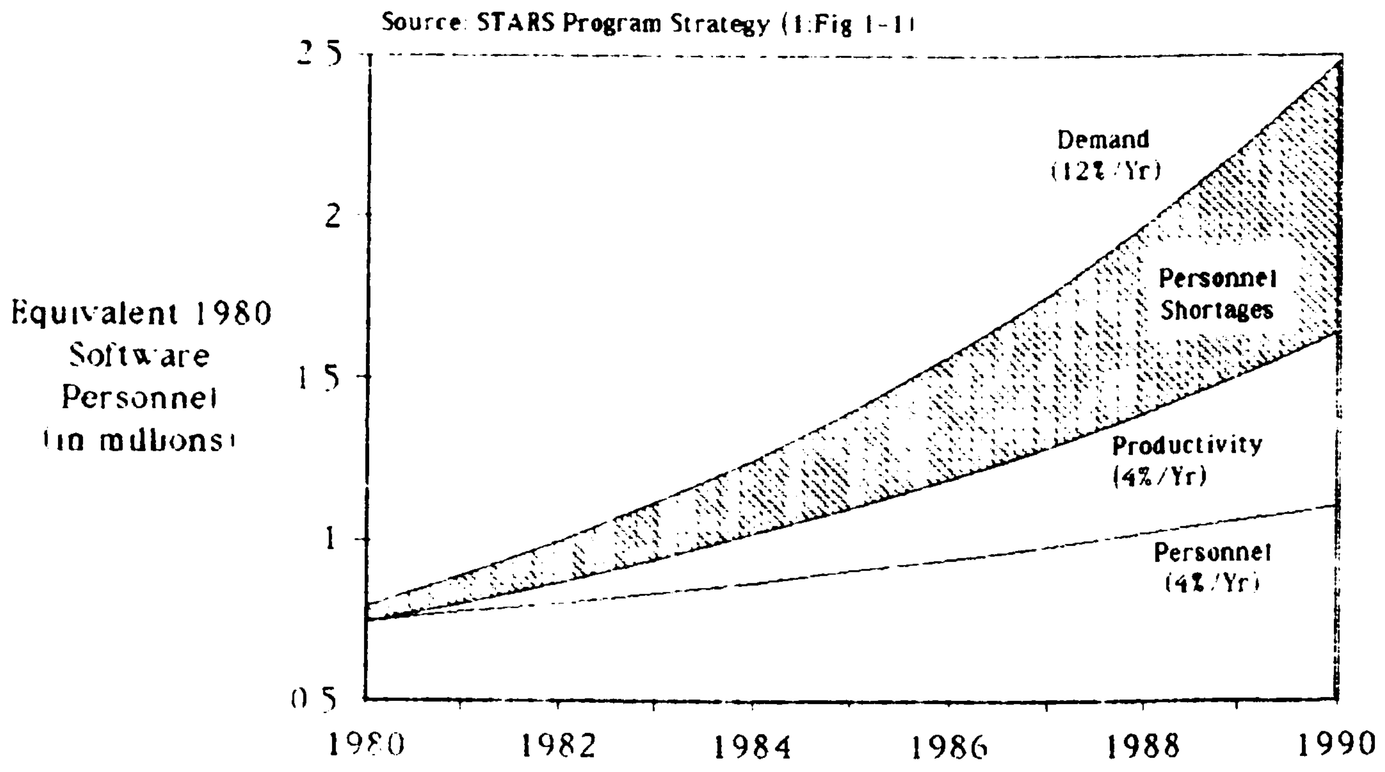


FIGURE 1. TRENDS IN SOFTWARE SUPPLY AND DEMAND

CHAPTER II

THE MODERNIZATION PROBLEM

"Although recognized since the 1960s, the software crisis continues to be the bottleneck in technological advancement." (2:vii) This bottleneck is due primarily to the labor-intensive nature of software programming. This labor intensiveness arises from, among other factors, the lack or inadequacy of:

- Computer programming language standards,
- Software portability,
- Automated support to programmers,
- Documentation of software, and
- Software architectures for applications.

The impact of these factors is felt throughout the software life cycle. The first effect is on software development; however, the impact carries through to the maintenance phase and establishes the basis of subsequent software modernization problems. There are also non-technical factors which impact heavily on modernization efforts.

Although the software problem is generally recognized, progress has failed to reverse the overall trend. This chapter will look at some factors contributing to this problem, and Chapter III will look at some ongoing efforts to address these factors.

Standards. There are literally hundreds of different software languages and dialects in use. A study conducted in 1978 indicated there were more than 450 languages used in the DoD. (4:209) These different languages limit software from being readily moved ("ported") from one system to another for reuse. They also generate excessive training requirements and limit flexibility in applying personnel resources.

Originally, software languages were machine-specific. That is, machines or product lines tended to use unique languages. To address this problem, standard high-order languages (e.g., COBOL, FORTRAN, JOVIAL) were developed. These languages are intended to be more like natural languages and machine-independent (i.e., portable between different makes of machines). Languages are often developed for specific classes of applications (e.g., business, scientific, military command and control). Generally, even "standard" languages are not portable between different machines.

> "The differences between various versions of the same language are serious impediments to software portability. Dialect differences are still considerable in two major languages that were originally standardized in the 1970s: COBOL and FORTRAN." (5:19)

Some language implementations are subsets (i.e., partial implementations) and others are proprietary supersets (i.e., unique extensions to the language). This latter case is particularly troublesome since "... a good marketing strategy was to include nonstandard features in the hope that users would use them and thus get locked into the manufacturers' hardware." (5:7) The most critical step in achieving software portability is to develop and implement a standard language. (5:6)

Portability. Unless software can be directly ported from one machine to another, common applications and routines must be at least partially recoded for the various machines. Depending on the software's complexity, this may demand significant time and effort. This wastes critically short manpower and is prone to inducing errors in the software. At best, identifying and correcting these errors are time-consuming; at worst, errors may go undetected until the software is in operational use.

These problems are compounded every time a change is to be made in the software, as it must be propagated through the various implementations. And, as previously pointed out, change is an inherent quality of all software.

Automated Support Tools. One of the key methods of reducing the labor-intensiveness of programming is the use of automated support tools. Large software programs are complex systems with many interactive parts. Design and implementation of large software systems require the concurrent ef[illegible]t of one or more teams of programmers. Further, during development, the elements of the overall program exist in different forms and at varying levels of abstraction (e.g., concept, top-level design, detailed design, source code, object code). This complexity tends to give rise to programming errors.

Almost all errors which occur during software development and maintenance are human errors. In fact, most programming tasks are inherently error-prone if executed by people. (6:183) This is because there is virtually no tolerance for error. Consequently, the work requires extreme concentration and constant, painstaking attention to detail. These traits are hard to sustain for long periods of time.

While most errors are found during development and maintenance and are remedied without operational impact, they still result in considerable amount of rework. This severely limits the programmer productivity which can be achieved.

Clearly, to enhance programmer productivity, it is desirable to automate these error-prone tasks to the maximum extent practicable. For even modestly-sized software activities, the costs of automated support tools are rapidly amortized through increased productivity. In addition, they help preclude or eliminate errors which would otherwise go undetected until after the system were operational. This provides higher quality in operational software and second-order productivity improvements by avoiding some of the maintenance activity which would subsequently be required once the software were in operational use.

Table 1 lists some of the automated support tools which are available. It is beyond the scope of this paper to explain the specific functions of each tool; however, they can provide critical support and productivity enhancements throughout the software life cycle.

Automated support tools is another area where lack of standardization has caused problems. The command languages used to control these tools, as well as the functionality which they provide, has tended to be machine- or product line-dependent. As in the case of multiple programming languages, this results in additional and costly training requirements and limits the ability to move programmers between systems. Further, limited resources are available for automated support tools. Consequently, full complements of automated tools for multiple systems may not be affordable. For computers with a relatively small commercial base, the full range of tools may not even be developed.

Table 1. **Automated Support Tools***

Assemblers	Interpreters
Code Generators	Linkers
Command Language Processor	Loaders
Compilers	Prettyprinter
Configuration Control Tools	Runtime Support Libraries
Data Base Management System	Set-Use Static Analyzer
Diagnostics	Stub Generator
Emulators and Simulators	Symbolic Dynamic Debugger
File Administration Tools	Text Editor
File Comparator	Text Formatter
Frequency Analyzer	Timing Analyzer

* (6:Table 8-1; 7:Fig 21.1; 8:Table 2)

Documentation. Accurate and detailed documentation of a program throughout its life cycle is essential. The ability to efficiently develop and maintain a software system is dependent first and foremost on the programmer's understanding of the system's purpose, structure, and methodology. This can be a very difficult task. If any appreciable time has elapsed since initial development, even one of the program's original programmers typically has difficulty in fully understanding and working with even modestly-sized programs.

Modern programming languages are structured to encourage, and in some situations force, programming techniques which provide a degree of self documentation. Further, many of the automated support tools mentioned above are specifically intended to facilitate documentation and configuration control of software. However, there are two serious problems.

First, many operational software systems were developed using older, less structured programming languages and without benefit of the newer, more powerful automated support tools. Since approximately two-thirds of all software activity is maintenance, these older, less efficient languages tend to be self-perpetuating. (5:18)

Second, as software changes in an operational environment, both its logical structure and documentation tend to deteriorate. Often this is caused by pressure to restore a system to operation or provide urgently needed additional capability as fast as possible. Quick becomes the enemy of clean. These operational patches occasionally inject new errors, which are themselves hastily patched.

> Because most software has not been designed to tolerate change and software engineers have not been taught to anticipate and prepare for change, software quality deteriorates as a negative side-effect of change, especially during the maintenance phase. (2:viii)

Software Architectures. One of the primary means of reducing software manpower demands is to reuse existing software. However, seldom does software for one application exactly match the validated requirements for another application. This limitation can be minimized by "cutting and pasting" modules from existing software and reusing these modules or packages. The use of structured programming, top-down design, and step wise refinement significantly enhances this capability. Under this methodology, programs are built of discrete modules. In the ideal situation, each module becomes a "black box." To implement its function, a module accepts an input and transforms the input to produce an output (data or action). The mechanics of this transformation are invisible to elements of the program outside of the module. All that is seen is the functional behavior of the module and its input and output interface characteristics. For the various modules to work together, it is necessary to specify exactly their interfaces, that is, each module's input and output characteristics. The functional allocation and interface specifications constitute the major portion of the top level design and establish an architecture for the overall software program.

If two programs perform essentially the same function but have different architectures, the functional allocation and interface characteristics of their component modules will be different. Even slight differences will make use of one program's modules in the other impossible without some rework. The level of effort required by this rework may be significant. Consequently, standardized architectures for various applications are necessary to enhance sharing of software modules. Unfortunately, this has not been accomplished generally. However, much work has been done along this line for the software which implements data communications protocols for packet switching networks. (9:Annex C)

Non-Technical Factors. While the factors outlined above significantly contribute to the software modernization problem, there are many non technical, managerial factors which are also significant: (17:1-2)

- Attitude – emphasis is on development of new applications rather than modernization of existing applications.
- Historic precedent – inclination to continue use of in-house development capabilities rather than purchase modern commercial software.
- Difficulty of technology transition – cost, training requirements, and resistance to change contribute to short-term "optimization" over longer-term benefits.
- Uncertainty – competing and rapidly changing technologies raise fears of selecting a "wrong" or immature technology.
- Lack of life cycle costing – long-term impact of software modernization is difficult to quantify and substantiate.

These technical and non-technical factors must be dealt with to insure that mission support can be provided and sustained. The next chapter addresses some of the major ongoing approaches to modernize existing and future software systems

CHAPTER III

ONGOING APPROACHES

In the previous chapter some of the factors which make software programming labor-intensive and which impact on software modernization were outlined. This chapter looks at some ongoing efforts to reduce this labor-intensiveness and provide for software modernization.

Software Technology for Adaptable, Reliable Systems (STARS). The STARS program was initiated to improve the United States ability to "exploit the advantages of computer technology through software." (1:viii) It is intended to rapidly improve the "state of practice," that is, advance both the technology base and the level of technology in daily use. The program's efforts are directed across the spectrum of acquisition, management, development, and support of computer software used for military systems. The goal of the STARS program is to improve software productivity while achieving greater system reliability and adaptability. The program's objectives are as follows: (1:x-xi)

- Improve the personnel resource by:
  - increasing the level of expertise,
  - expanding the base of expertise available to DoD;
- Improve the power of tools by:
  - improving project management tools,
  - improving application-independent technical tools,
  - improving application specific tools;
- Increase the use of tools by:
  - improving business practices,

- improving usability,
- increasing the level of integration,
- increasing the level of automation.

The STARS program also initiated efforts to establish a Software Engineering Institute (SEI). However, the SEI was subsequently established at Carnegie Mellon University, Pittsburgh, Pennsylvania, as a separate program. The SEI is to translate technology base improvements from research and development (R&D) activities to operational applications. To assist in this effort, the SEI will maintain a state-of-the-art software environment testbed.

<u>Ada Programming Language</u>'. As pointed out in Chapter II, the proliferation of programming languages results in excessive costs in software development, maintenance, and training; hinders software modernization; and makes it difficult to move programmers from one project to another. Many of these languages and dialects and their associated support environments (e.g., compilers) have not been formally or thoroughly tested and proven. Their use involves a degree of technical risk. (4:209) These risks are of particular concern in DoD applications where software can affect weapon systems performance, safety of flight, information security, etc.

In 1975, the DoD initiated an effort to develop a state-of-the-art high-order programming language and support environment. This language was to be machine-independent (i.e., truly portable) and tightly controlled to preclude implementations of either subsets

'Ada is a trademark of the US Department of Defense or supersets of the language. The resulting language, Ada, is to be the single standard programming language for embedded computers and other mission-critical applications. (10:1) It will be used when developing standard software systems for both mission-critical and non-mission-critical, general-purpose information processing applications (i.e., automatic data processing or ADP). However, some specialized languages will continue to be used indefinitely in applications where Ada cannot meet validated technical or performance requirements.

In addition to providing a standard, portable, state-of-the-art, high-order programming language, the Ada program has a formal validation process for Ada compilers. This validation process is partially enforced by controlling the use of the name "Ada." The DoD obtained a copyright on the name to insure that it is only used for DoD-validated compilers. Ada will also have a standard programming support environment (i.e., portable automated software tools).

Some estimates have indicated that Ada has the potential to save the DoD several million dollars per year by lowering training costs, increasing programmer productivity, and enabling substantial reuse of standard code modules. (3:78)

<u>End-User Computing</u>. As pointed out in the Introduction and shown in Figure 1, software demand far exceeds the supply of qualified software programmers. As unfavorable as this projection is, some experts have projected an even more unfavorable supply to demand ratio, with demand exceeding supply by a factor of four by

1990. However, this is not just a problem for the future, software design centers are already "hopelessly overtasked with years of backlog." (11:17-18)

There are a few technology trends which can help to alleviate this problem:

- Rapidly falling hardware costs,
- Proliferation of relatively inexpensive but powerful mini- and micro-computers, and
- Powerful commercial fourth-generation programs which provide nonprocedural "user-friendly" interfaces.

These trends are supporting the decentralization of information processing into functional workareas (e.g., operations, administration, personnel, logistics). This decentralization is driven by two related factors. The backlog of development work has made the software design centers and information systems facilities unresponsive to all but the highest priority work. In "self defense," functional area managers have had to look to their own support through the application of small computers. On the other hand, the information systems professionals have had to encourage this movement to alleviate some of the excessive demand on their resources.

For relatively straightforward "business-like" applications (e.g., word processing, spreadsheet analysis, business graphics, small data base applications, terminal communications), commercial software packages can provide superb and efficient capabilities (albeit, with widely varying quality, levels of sophistication, and ease of use)

These packages can meet many of the end-user requirements with little development or maintenance. However, these applications are not at the core of many functional areas.

End-user computing has the potential to provide additional assistance in addressing the real functional area automation requirements. One of the major time factors and problem areas with developing application software is in first developing an adequate understanding of the application to be automated. Statements of requirements are noted for their ambiguity, lack of adequate detail, and continually evolving nature during the development process. Professional programmers often embark on a development without any experience in the functional area which they are to help automate. This requires a learning curve just to reach a point where the proper questions can be asked of the functional area users. With end-user computing, the functional experts can interact directly with the program throughout the development. This can reduce the development time and produce a higher quality output. It should at least produce what the end-users need, as opposed to what they said they needed.

Data Base Management Systems (DBMSs). In looking at the portability of software between systems, the portability of the associated data must be addressed. The importance of this issue is reflected in the fact that a large proportion of the computer resources of a typical data processing installation is spent in sorting and merging data. Some estimates place this usage as high as 25 percent of the total resources. (5:10)

The problem of portability of data is compounded by such factors as the use of different nonnumeric collating sequences by some compilers. In addition, an efficient data storage scheme for one DBMS might be "disastrous" for another. (5:10)

There are several interrelated approaches to address this problem:

- Relational DBMSs (R-DBMSs)
- Data base machines
- Standardized DBMS query languages

Relational DBMSs provide very flexible data structures which are easier to use. This provides for lower development and maintenance costs for application programs which make use of the DBMS. However, this flexibility comes at the cost of either reduced speed performance (responsiveness) for given computing power (processing and memory capacity) or, alternatively, a requirement for increased power for a given performance level. A highly effective method of providing the extra computing power is to off-load the data management functions to a dedicated procesor optimized to perform this function, i.e., a data base machine. However, to achieve the full benefits of these capabilities, it is necessary to standardize the application programs' interface to the DBMS (in this case, the R-DBMS). This is the function and purpose of a standardized query language such as the Structured Query Language. (12:Atchs 2, 3)

To address these issues,

the Air Force is working with the Navy and the Army to establish a joint requirements contract for a data base machine. The contract will include high level specifications (including relational model [an R-DBMS] and Structured Query Language (SQL) interface). (12:1)

Further, the Air Force is working towards the adoption of SQL as the Air Force standard language for use with R-DBMSs. (12:2)

As seen throughout this chapter, STARS, Ada, end-user computing, modern DBMSs, and other ongoing efforts are attacking many of the fundamental problems. However, additonal emphasis is needed in other areas to effectively address software modernization Chapter IV will identify some of these other approaches.

CHAPTER IV

RECOMMENDATIONS

Chapter III identified several ongoing efforts to reduce the labor-intensiveness of software programming and to provide for software modernization. However, these are primarily long-term efforts and must be supported by more fundamental changes in attitude and approach toward dealing with software modernization. This chapter will address some complementary actions which can be instituted for both near- and long-term enhancements.

People Efficiency Over Machine Efficiency. Initially, computers (i.e., hardware) were very expensive, had relatively limited power, and were not widely available. As a result, all aspects of computing were oriented towards maximizing the efficient use of the computer hardware system. Even as the cost of hardware fell and computing power and availability increased, computer applications expanded as rapidly. Processing power and time remained a critical limiting factor. This relationship still exists today in some areas; for example, the continual pressure for ever more powerful:

- Supercomputers for applications which are computationally complex, and
- Small microprocessors for real-time applications which are weight and space limited.

However, there is a fundamental change progressively taking place—the critical limiting factor is increasingly software and, more specifically, qualified software programmers. Over the last decade hardware costs have decreased by two orders of magnitude (factor of 100) while programmer productivity has only increased by an order of magnitude (factor of ten). (2:3)

With this shift in relative importance from hardware to software, a corresponding change in attitude toward system acquisition is required. For example, the Air Force Phase IV program is a *capital replacement* program for the standard base level computer systems (UNIVAC 1050-II and Burroughs 3500/3700/4700). The strategy for the Phase IV program was (13:14)

- To rigidly control the software functional baseline of the standard base-level applications as of 1980;
- To replace obsolete UNIVAC and Burroughs hardware with the new Phase IV environment;
- To convert the software functional baseline to run within the Phase IV environment and apply *limited updates as necessary*; [emphasis added]
- To do all this without loss of functional capability or degradation of response time

While this conservative strategy minimized system conversion risk, it failed to adequately address software modernization. In particular, it failed to recognize that modern software tools are resource-intensive (memory and processing power) and to analyze the life cycle impact of continuing the functional baseline versus a strategy designed to maximize user and programmer productivity. Such an analysis must address opportunity costs associated with dwindling software development capabilities unless significant productivity enhancements are made. The required productivity enhancements can only be achieved if there is a fundamental change in future system designs corresponding to a change in strategy from machine efficiency to people efficiency.

Further, with the shift toward end-user computing, there must be a radical change in the method and demands of the man-machine interface (this user interface includes the interface to the operating system, application programs, and hardware). End-users are functional experts, not computer experts. To the extent that they must become computer experts to make effective use of their decentralized automated support, their efforts are diverted away from their functional work. This has the potential to offset much of the productivity gains which the automated support was intended to provide.

Consequently, the user interface must be intuitive and conform to the users' conceptual model for the application, not a computer systems designer's model of the application. (14:3) While advances such as application-specific fourth generation languages (e.g., advanced relational data bases, powerful and flexible spreadsheet applications) provide significant help along this line, much remains to be done—particularly in terms of the technology fielded vice what is available. Techniques such as event-driven operating systems and applications (vice rigid sequential and procedural designs), visually-oriented and voice input and output, extensive input edits, and the ability to interactively handle and resolve contextual ambiguities, are required to reduce the demands on the users. Progress has been made in many of these areas, however, most of the systems provided to the end-users fail to make adequate use of these technologies. In part, this is because systems tend to be specified by information systems specialist who by virtue of their technical training and experience find the existing interfaces logical and quite useable, even "friendly." However, this tolerance for these interfaces is extremely costly in demands on end-users who typically lack the necessary training and experience.

Although end-users have typically been able to make effective use—although sometimes quite limited—of currently fielded systems, this is due in part to the relatively limited application of these systems and extensive use of local "gurus" who have a strong personal interest in the computer systems. However, as more and more of the workload shifts to the end-users, this will become a progressively greater problem. Further, the likelihood of adequately sustaining this method of support under the stress of combat is low. The time and personal expertise available under stress could fall off precipitously.

Fewer and Better Trained People. The growing shortage of qualified software programmers relative to the increasing demand requires a more capital-intensive approach to meet validated mission requirements. While in principle, the laws of supply and demand would indicate that sufficient supply (of personnel) could continue to be met at progressively greater costs, there is a more fundamental constraint. The military services are limited both economically and by law on their military and civilian personnel strengths. In fact "projected manpower shortfalls across the Five Year Defense Plan (FYDP) threaten the ability of the Air Force to man new weapon systems as they are deployed." (11:1) Clearly, such direct combat missions have a higher priority for personnel resources; and mission support areas must find ways of sustaining, and even expanding, support within *or below* current personnel levels.

While ongoing efforts to increase the use of commercial software, contracted services, and end-user computing will help address this problem, more must be done. Higher levels of investment in automated support environments must be made. These investments will be cost effective from a cost-avoidance standpoint, but will probably not result in large personnel reductions on their own, particularly in the near-term. However, the issue is increasingly becoming one of adequate mission support vice cost-effectiveness.

Further, to make full use of, and thereby derive the full benefit of, automated support environments, significant additional training is required to update the technical knowledge of in place personnel. "The skill levels of the human resources have been identified as the most important single influence on software productivity." (1:53) This additional training is required both for in house development, as well as, to adequately specify, contract for, and monitor contractor services.

As the pressure to reduce personnel increases, programmer productivity must become a critical performance factor. The limited affordability of in house personnel will necessitate increased efforts to identify individuals with high productivity potential and to eliminate others during their probationary periods. This will result in additional turnover, higher training demands, and significant stress on supervisory personnel to make and enforce these judgments on individual performance and potential. In order for the supervisory personnel to accurately make these determinations, they also will need additional technical training to maintain knowledge of state of the practice and state of the art methods and technologies.

Interoperability and Networking. The Ada programming language offers significant potential to reduce personnel requirements through the reuse of software modules and packages. However, there are several practical limitations to extensive software reuse. Some of the major factors are the level of awareness of the availability of a given software module and a rapid means for obtaining the module when required.

Current procedures of the Federal Information Resources Management Regulation (FIRMR 201-31.014) require submission of abstracts for common use programs and systems to the Federal Software Exchange Center for publication in the *Federal Software Exchange Catalog* (15:ii). However, this document is published infrequently (annually) and does not provide for rapid, responsive dissemination of programs which appear useful. What is required is an on-line catalog with networked access and a means of rapidly telecommunicating program modules from one computer installation to another.

While the Ada program addresses establishment of support program libraries (16-18), general availability of program modules from program library requires functional interoperability of computer systems on a wide-scale basis and extensive networking. The Defense Data Network (DDN) offers the means of networking the resources, however, full recognition of the necessity for wide-scale interoperability has not been achieved. Interoperability is a basic requirement which transcends the immediate operational requirement which may not directly necessitate full interoperability. Until wide-scale functional interoperability is achieved, effective software reuse will be unnecessarily limited.

Policy Approach. The recommended changes outlined above require basic changes in attitude and approach in addressing software modernization issues. The inertia inherent in people and organizations which results from resource constraints, communications barriers, fear of change, etc., opposes efforts to effect the required changes. Firm policy must be established to guide and motivate organizational direction.

Policy must frequently precede an organization's capability to effectively implement the policy. Policy should be prescriptive rather than descriptive—guiding rather than documenting the organization's direction. This is an essential element of leading.

To be effective, the policy must also be enforced. The basic problem is that if the policy is too far ahead of the organization's ability to implement it, then exceptions to the policy will dominate and the policy will be meaningless. However, unless significant pressure is applied to conform to a progressive policy, reasons to postpone its implementation become an easy route to avoid the management problems associated with making required changes.

The growing problems with software modernization necessitate that a progressive policy be established and enforced. The short-term impacts must be faced because the long term costs of failing to act would be nothing short of failing to support mission requirements.

CHAPTER V
CONCLUSION

The dependence of the United States on qualitative superiority to maintain deterrence has placed increased emphasis on software for flexible and responsive support of mission-critical and general-purpose applications. However, the responsiveness and integrity of software systems is being jeopardized by the labor-intensiveness of software programming and the growing shortage of qualified software programmers relative to the expanding demand.

Although the software problem is generally recognized and numerous initiatives have been started to improve the situation, progress has failed to reverse the overall trend. The technological approaches to improve the productivity of software programmers, enable reuse of software packages, and accelerate technology transition and integration into the state-of-the-practice, offer significant long term potential to reverse this situation. However, near-term gains are limited by existing attitudes and approaches toward software modernization. Fundamental changes are urgently needed to minimize near term problems and accelerate the achievement of long term benefits.

The most critical near-term step in reversing the negative trends is the establishment of progressive and demanding policies on software modernization. This is essential to continued support of US strategic and tactical capabilities.

LIST OF REFERENCES

1. *Software Technology for Adaptable, Reliable Systems (STARS) Program Strategy*. Washington, DC: Department of Defense, 15 March 1983.

2. McClure, Carma L. *Managing Software Development and Maintenance*. New York: Van Nostrand Reinhold Co., 1981.

3. *Information Technology R&D: Critical Trends and Issues*. Washington, DC: U.S. Congress, Office of Technology Assessment, OTA-CIT-268, February 1985.

4. Elson, Benjamin M. "Software Update Aids Defense Program," *Aviation Week & Space Technology*, March 14, 1983, pp.209-221.

5. Wolberg, John R. *Conversion of Computer Software*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

6. Dunn, Robert and Ullman, Richard. *Quality Assurance for Computer Software*. New York: McGraw-Hill, 1982.

7. Saib, Sabina. *Ada: An Introduction*. New York: Holt, Rinehart and Winston, 1985.

8. Wolf, Martin I. et al. "The Ada Language System," *IEEE Computer*, June 1981, pp. 27-35.

9. *Air Force Information Systems Architecture, Volume I, Overview*. Washington, DC: Headquarters United States Air Force, 8 May 1985.

10. Interim DoD Policy on Computer Programming Languages Memorandum, USDR&E to Secretaries of the Military Departments et al., 10 June 1983.

11. *Report on Data Systems Management and Manpower Impacts, Volume I, Executive Summary.* Washington, DC: Air Force Management Analysis Group, 1 September 1984.

12. Database Machine Requirements Contract and Standardization on Structured Query Language (SQL). Letter, Headquarters United States Air Force/SIT, to all Major Command/SI et al., 18 June 1985.

13. *Interim Report on Air Force Base Level Automation Environment.* Washington, DC: National Academy Press, January 1986.

14. Sime, Max E and Coombs, Michael J (ed). *Designing for human-computer communication.* London: Academic Press, 1983.

15. *Federal Software Exchange Catalog.* Washington, DC: National Technical Information Service, PB85 904001, 1985.

16. Stenning, Vic et al. "The Ada Environment: A Perspective." *IEEE Computer*, June 1981, pp. 17-26.

17. Software Modernization. Headquarters United States Air Force/SITT staff paper, undated (c. 1985).

GLOSSARY

AB	Air Base
Ada	Department of Defense standard high-order computer programming language. Named in honor of Augusta Ada Byron, Countess of Lovelace (1815-51), the first computer programmer. (ANSI/MIL STD 1815A 1983)
ADP	Automatic Data Processing
AFB	Air Force Base
AFIT	Air Force Institute of Technology
ANSI	American National Standards Institute
BASIC	A high-order programming language intended for use by minimally trained personnel to do relatively simple programs. (Beginners' All-purpose Symbolic Instruction Code)
B.S.	Bachelor of Science (degree)
Burroughs 3500/ 3700/ 4700	Standard Air Force base level computer prior to Phase IV capital replacement program.
COBOL	A high-order programming language for business applications. (COmmon Business Oriented Language)
DBMS	Data Base Management System
DDN	Defense Data Network
DoD	Department of Defense

F 111	Long-range interdiction fighter aircraft
FIRMR	Federal Information Resources Management Regulation
FORTRAN	A high-order programming language for technical and scientific applications. (FORmula TRANslator)
FSEC	Federal Software Exchange Center, 5285 Port Royal Road, Springfield, VA 22161. (703) 487-4848.
FY	Fiscal Year (1 October – 30 September)
HQ	Headquarters
IEEE	Institute of Electrical and Electronics Engineers
JOVIAL	A high-order programming language for command and control applications. (Jule's Own Version of International Algorithmic Language)
MIL-STD	Military Standard
Mission-Critical	An application exempted from the Brooks Act by 10 U.S.C. 2315, the Warner Amendment to the FY 1982 Defense Authorization Act. (10:1)
M.S.	Master of Science (degree)
Phase IV	Program for capital replacement of standard Air Force base level computers (UNIVAC 1050-II and Burroughs 3500/3700/4700) with Sperry 1100/60 computers
Ph.D.	Doctor of Philosophy (degree)
R&D	Research and Development
R-DBMS	Relational Data Base Management System

SEI	Software Engineering Institute (Carnegie Mellon University, Pittsburgh, Pennsylvania)
SI	Assistant Chief of Staff for Information Systems (HQ USAF), or Deputy Chief of Staff for Information Systems (Major Command).
SIT	Architecture and Technolgy Directorate, Assistant Chief of Staff for Information Systems (HQ USAF)
SITT	Technology and Security Division, Assistant Chief of Staff for Information Systems (HQ USAF)
SQL	Structured Query Language
STARS	Software Technology for Adaptable, Reliable Systems
UNIVAC 1050-II	Air Force Standard Base Supply System computer prior to Phase IV capital replacement program.
US	United States (also written U.S.)
USAF	United States Air Force
U.S.C.	United States Code
USDR&E	Under Secretary of Defense for Research and Engineering

END

4-87

DTIC