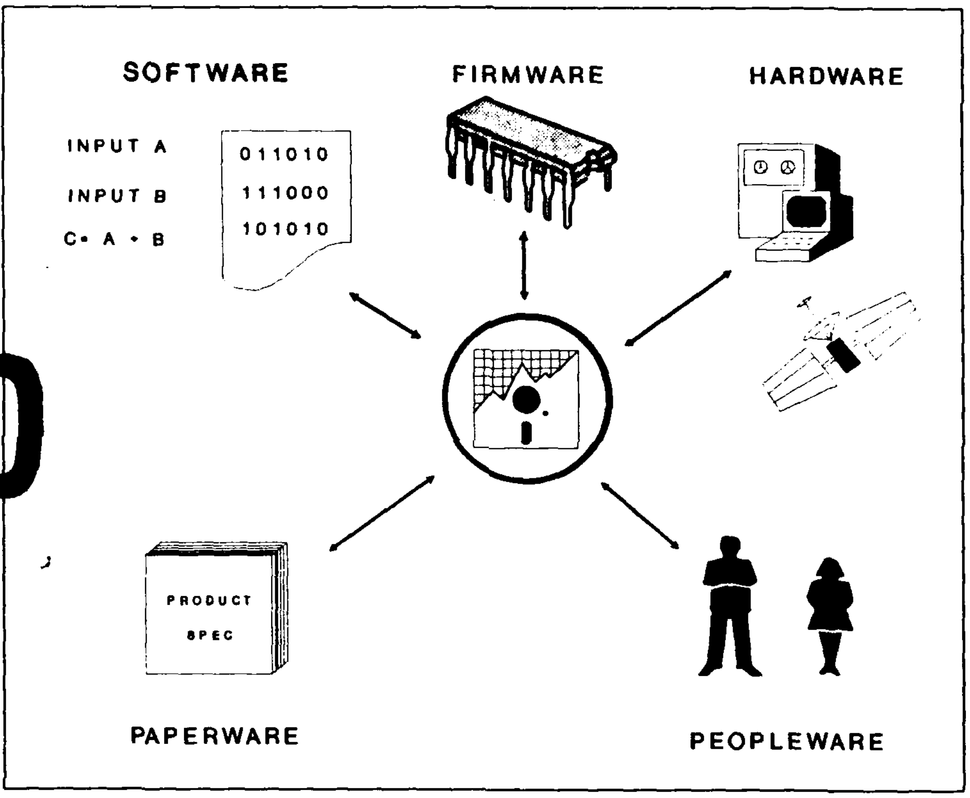
1

MISSION CRITICAL COMPUTER RESOURCES MANAGEMENT GUIDE



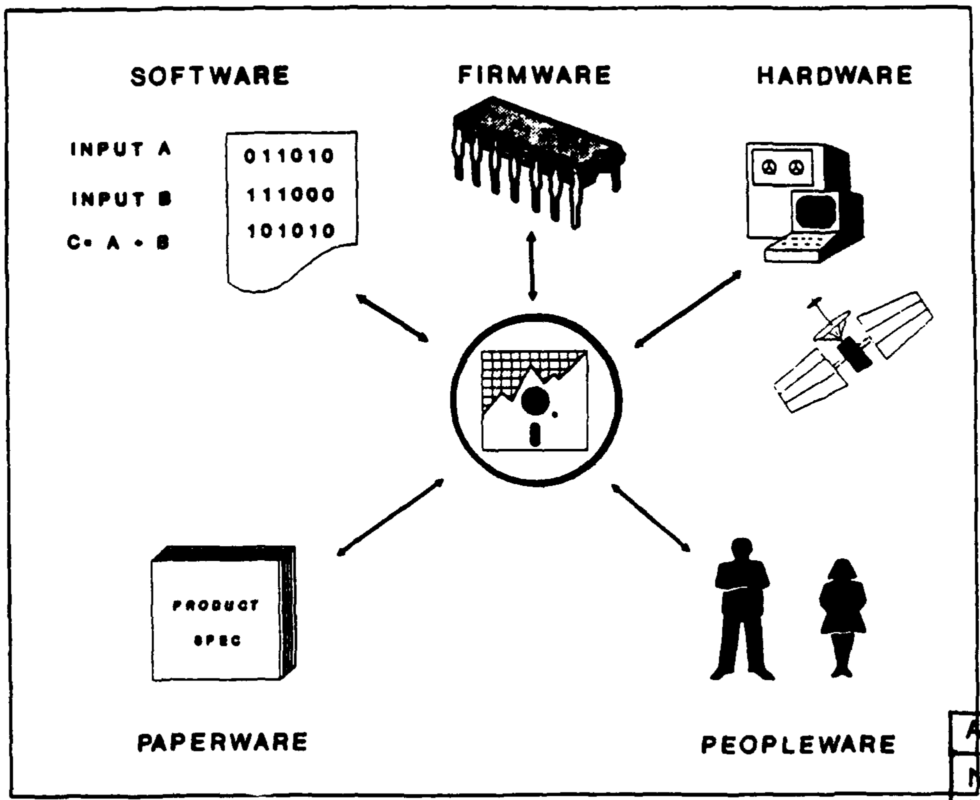
TECHNICAL MANAGEMENT

SEPTEMBER 1988

STATEMENT "A" per Lt. Caro
DSMC-SE-T
TELECON 2/6/90 CG

PRICE-$11.00 per GPO
TELECON 2/7/90 CG

MISSION CRITICAL COMPUTER RESOURCES MANAGEMENT GUIDE



TECHNICAL MANAGEMENT

SEPTEMBER 1988

For sale by the Superintendent of Documents, U.S. Government Printing Office
Washington, D.C. 20402

PREFACE

This document is one of a family of educational guides written from a Department of Defense (DOD) Perspective; i.e.,non-service peculiar. These books are intended primarily for use in the courses at the Defense Systems Management College (DSMC) and secondarily as a desk reference for program and project management personnel. The books are written for current and potential DOD Acquisition Managers who have some familiarity with the basic terms and definitions of the acquisition process. It is intended to assist both the Government and industry personnel in executing their management responsibilities relative to the acquisition and support of defense systems. This family of technical guidebooks includes:

- Integrated Logistics Support Guide; First Edition: May 1986
- Systems Engineering Management Guide; Second Edition: Dec 1986
- Department of Defense Manufacturing Management Handbook for Program Managers; Second Edition: July 1984

This guidebook was developed by the following members of the DSMC Technical Management Department staff:

Lt Col Israel I. Caro, USAF
Lt Col Ronald P. Higuera, USAF
Cdr Frank R. Kockler, USN
Mr. Sherwin J. Jacobson
Mr. Alan Roberts

TABLE OF CONTENTS

APPENDICES

CHAPTER 1
INTRODUCTION

Mission Critical Computer Resources (MCCR) refers to the totality of computer hardware and computer software that is integral to a weapon system along with the asscciated personnel, documentation, supplies and services. A natural question to ask is "Why should a program manager be that interested in MCCR. The answer is threefold. First, software for embedded systems is on the "critical path" of system development, so if software development falls behind schedule, the entire weapon system development will also fall behind schedule. Second, MCCR can produce development problems of sufficient magnitude to result in costly program overruns. It is nct uncommon for software development costs to exceed initial budget estimates by as much as 50% to 100%. Third, the performance of modern weapon systems is largely dependent on the quality of their computer resources; the system is only as good as its software. To further compound the problem, it is an established fact that if the MCCR development falls behind schedule, the development lead times cannot be shortened simply by applying more resources. Money can't fix the problem - only time can [1]. Once a program falls behind, little can be done to save it!

Management of MCCR development cannot be ignored or delegated. If the program manager leaves all MCCR management considerations to the development contractor, there is the strong possibility that the software development will encounter significant difficulties. The management of MCCR development may be compared with piloting an aircraft. Without the proper application of the necessary control, it is extremely unlikely that the aircraft will safely reach the intended destination. Without the appropriate management directives, it is unlikely that MCCR development will result in a suitable product. There are no autopilots for MCCR development. Effective MCCR acquisition management, like piloting, is a difficult task but, with proper knowledge and care, not impossible [2]. This guide will only cover the basics by providing enough background to enable straight and level flight. Aerobatics (i.e., unconventional developments) depend on greater mastery of the fundamentals, which are outside the scope of this guidebook.

The actual MCCR development will be accomplished by a development organization belonging to a system development contractor. On occasion, the "contractor" may be another DOD agency. The development contractor has the responsibility of delivering a software product that meets all contractual requirements. Unfortunately, it is not possible to precisely and completely specify in a contract all the characteristics of the final software product and its development process. Experience has shown that the difference between successful and unsuccessful development efforts is often the rigor and timeliness of the direction given to the contractor by the procuring agency's program management organization [2].

REFERENCES

1. Meinke, George H., "Airborne Software Acquisition Management...A Guide for New Software Managers", Air Command and Staff College Report Number 82-1685, Air University, Maxwell AFB, Al 36112.

2. Rubey, Raymond J., "A Guide to the Management of Software in Weapon Systems", 2nd Edition, March 1985.

CHAPTER 2
INTRODUCTION TO COMPUTER RESOURCES

2.1 BASIC DEFINITIONS

Before beginning a discussion of computer resources, it is necessary to provide some basic definitions. These definitions will not be exhaustive since they will merely serve as a point of departure for the initial discussions. As we progress through this guide, these definitions will become clearer as more definitive information is provided.

Autonomous

A system that operates without human intervention. An autonomous system, however, still requires initialization by a human operator. An example would be a Trident missile which requires a crew for launching but, once airborne, its trajectory and warhead delivery sequence is solely determined by on-board systems.

Digital Computer

An electronic machine that is capable of storing in discrete numeric form a predefined set of instructions, and all associated data, and subsequently executing these instructions.

Digital Systems

Any system which incorporates digital computers and software as integral functioning entities.

Embedded Computer Resources (ECR)

A computer system that is physically incorporated into a larger system (not necessarily within it) whose function is not data processing. It can be stand-alone, but it must still be integral to the larger system. The computer system can be used for other purposes provided the primary function is as part of the weapon system.

Large Scale

Systems requiring massive amounts of software usually in excess of one hundred thousand lines of instructions.

Man-in-the-loop

A system where a human operator must be an integral part in order for the system to function as designed. An example is an aircraft flown by a pilot.

Memory

The section of the computer that stores or holds all of the software.

Mission Critical Computer Resources

Any computer resource that is an integral part of a weapon system or an operational system and required by that system for proper operation. ECR is a subset of mission critical computer resources.

Real-Time

The immediate response to a stimulus. By immediate is meant on the order of milliseconds (i.e. thousandths of a second).

Software

The pre-defined set of instructions and associated data that are stored in a computer and are used to execute a function or functions.

Software Engineering

The *emerging technological* and *managerial* discipline dedicated to the development and support of computer software systems, on time and within cost estimates, through the creation and use of proven methods and automated tools [1].

Additional terms are defined in Appendix B.

2.2 HISTORICAL PERSPECTIVE

The development of computer software as a recognized activity is less than 30 years old. During the infancy of digital computers (1950's) all software or computer programs were developed by engineers or scientists as an adjunct to their work with computers. In the early 1960's, computer technology was usually taught under the auspices of university or college electrical engineering departments. It wasn't until the late 1960's that computer science departments were being established as separate entities. The term "software engineering" was not coined until 1968 when the term was used as a theme of several workshops held in West Germany and Italy to address the growing problems associated with software development [2]. So unlike other disciplines, such as electrical engineering, software engineering is a relative newcomer. Because of this, its practitioners do not have at their disposal the wealth of time-tested practices, procedures and tools so readily available to its sister disciplines. In circuit design, for example, an electrical engineer can use off-the-shelf components and modules with the necessary characteristics to build large portions of his system.

Off-the-shelf components or modules are not widely available to the software engineer. Quite often attempts to use existing software lead to major problems if the designers are not careful when integrating the existing software into their design. Designers must fully understand all of the characteristics of existing software as well as its overall performance and reliability. Software development is still in its infancy. New innovations are being introduced every day, but they are more evolutionary in nature than revolutionary. Many of the problems associated with software development are due to the relative immaturity of the discipline. There is still too much art and craft and not enough engineering in software development; although the trend is definitely changing.

The introduction and growth of digital systems in the DOD parallels the introduction and growth of digital systems in the commercial market place. In fact, the stringent requirements of military systems often spearheaded the development of computers and software throughout the industry, especially during the early years of computer development. In the early 1950's all weapon systems were analog and it wasn't until the mid-1950s that digital systems were introduced into weapon systems. During the 1960s there was a rapid incorporation of digital systems with an almost exponential growth occurring in the 1970s. Some of the reasons for this rapid growth were:

(a) Advances in integrated circuits (ICs), the basic building blocks of electronic equipment and digital computers. ICs were being developed with ever increasing capabilities and an accompanying decrease in power requirements, size and cost.

(b) The introduction of the microprocessor, which is essentially a computer on an chip, allowed designers to replace many pieces of hardware with a single component roughly the size of a postage stamp.

(c) The ever increasing Soviet threat and the need to counter it in the face of decreasing defense budgets. This drove the services to build fewer but smarter and technically superior weapons relying more and more on computers and software.

(d) The realization that software is inherently more flexible than hardware and better able to accommodate the ever changing threat.

(e) The tremendous advances made in the commercial marketplace in computers and software.

Today, all weapon systems are dependent on computers and software. This phenomenal growth of digital systems in aircraft, for example, is shown in Figure 2-1. In 1965 the FB-111 required an on-board computer memory of roughly 10,000 words but by 1980 the B-1B Bomber was approaching on-board computer memory requirements of about one million words. Current and future systems will greatly exceed these memory requirements with large scale software systems being the norm.

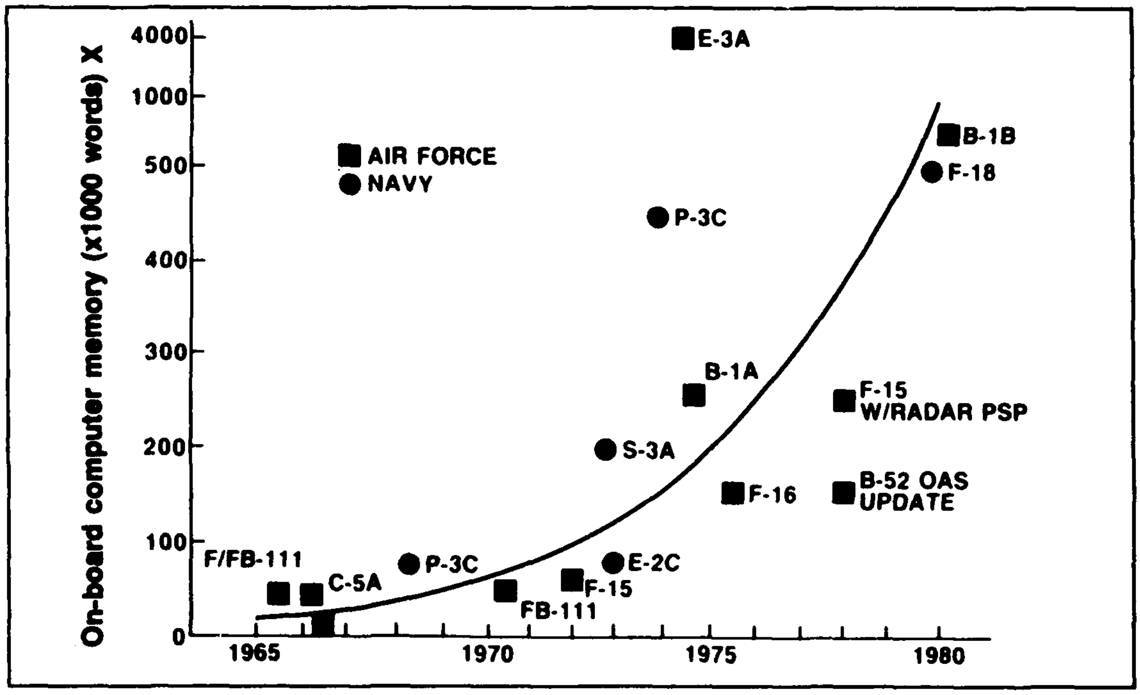


Fig. 2-1 Growth of Aircraft Digital Systems

2.3 SOFTWARE IMPACT ON SYSTEM

What does the curve in Figure 2-1 tell us? If one were to use the analogy of constructing a brick wall, one can say that a 3000 sq. ft. wall requires about three times as many bricks as a 1000 sq. ft wall. It is not necessarily more difficult to build, it just takes longer. Unfortunately this analogy breaks down when it comes to software.

The impact of software on system design and development is illustrated in Figure 2-2 [3]. The solid lines represent the average influence of either software or hardware on system design and development while the dashed lines on either side represent the maximum and minimum range of influence. It is very clear that in 1950 software had no influence on weapon systems design. This is because these systems contained no digital hardware. By 1980, however, the relative influence of software on system design averaged about 50% with some systems being influenced by as much as 70% or as little as 30%. This means that software considerations affected overall system design and development about 50% of the time. System engineers could no longer make hardware design decisions without considering the software implications. As can be seen, the trend seems to be for an ever increasing role for software. What the figure shows is that software is no longer merely a part of the system; software has become a system in its own right and has assumed the integration function for the various subsystems of a weapon.

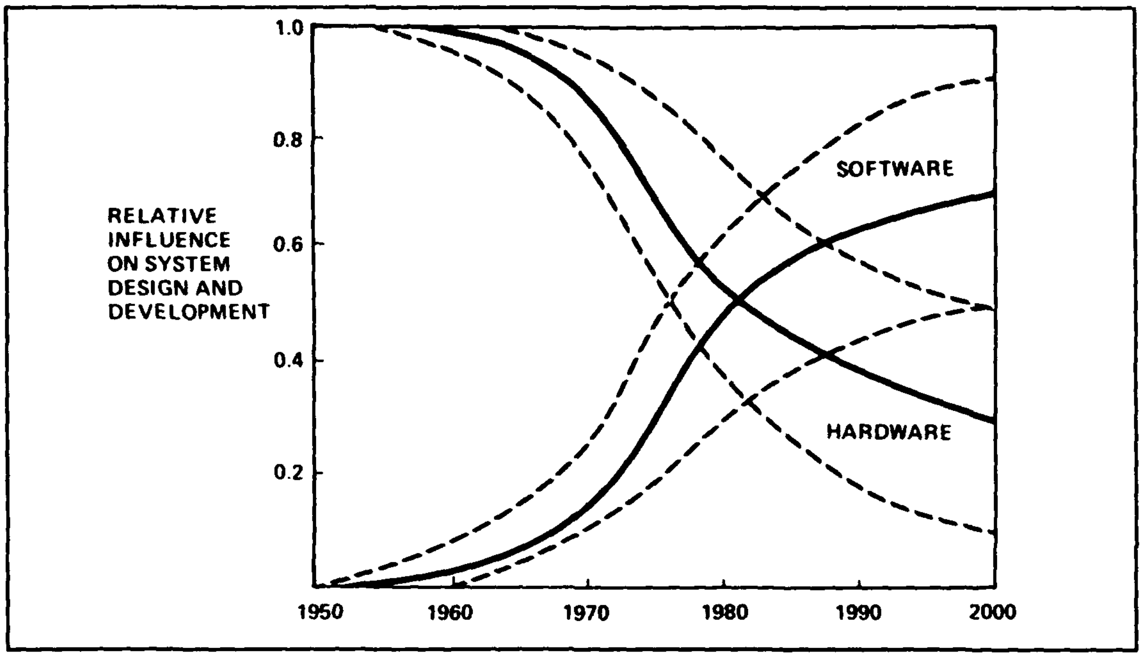


Fig. 2-2 Impact of Software on System Design and Development

2.4 LIFE CYCLE COST TRENDS

Figure 2-2 implies that hardware has been traded for software. But why should that be a problem? The problem is one of cost as shown in Figure 2-3 [4]. This figure shows that over the last 30 years the cost of the software, as a percentage of total computer resources cost, has been growing by leaps and bounds whereas the associated hardware has been decreasing by an equally dramatic percentage. A personal computer (PC) of today has more power than the large computers which were the workhorses of early space programs and they cost a mere fraction of their predecessors. In addition, a PC will sit on a corner of your desk while the early computers occupied a large room and required thousands of watts of electricity and tons of air conditioning.

The decreasing size of computer hardware, along with their ever increasing capabilities, has resulted in an explosion of applications in our weapon systems. Software not only performs many of the functions previously performed by specialized hardware, it also performs many of the functions which would be impossible or impractical to perform with just dedicated hardware. But this tremendous use of computers has come at a price. As Figure 2-3 shows, the ratio of hardware and software expenditures has changed from a ratio of 80% hardware and 20% software in 1960 to a ratio of 20% hardware and 80% software in 1980. But there has been an equally dramatic increase in the costs associated with supporting the software once the system is delivered. The primary reason why software development and support is so expensive is that both functions are extremely labor intensive. It is an ironic fact that an industry which has provided the means for other segments of industry to automate, has itself failed to automate. There are no machines and no computer programs that will automatically generate computer programs directly from a set of requirements.

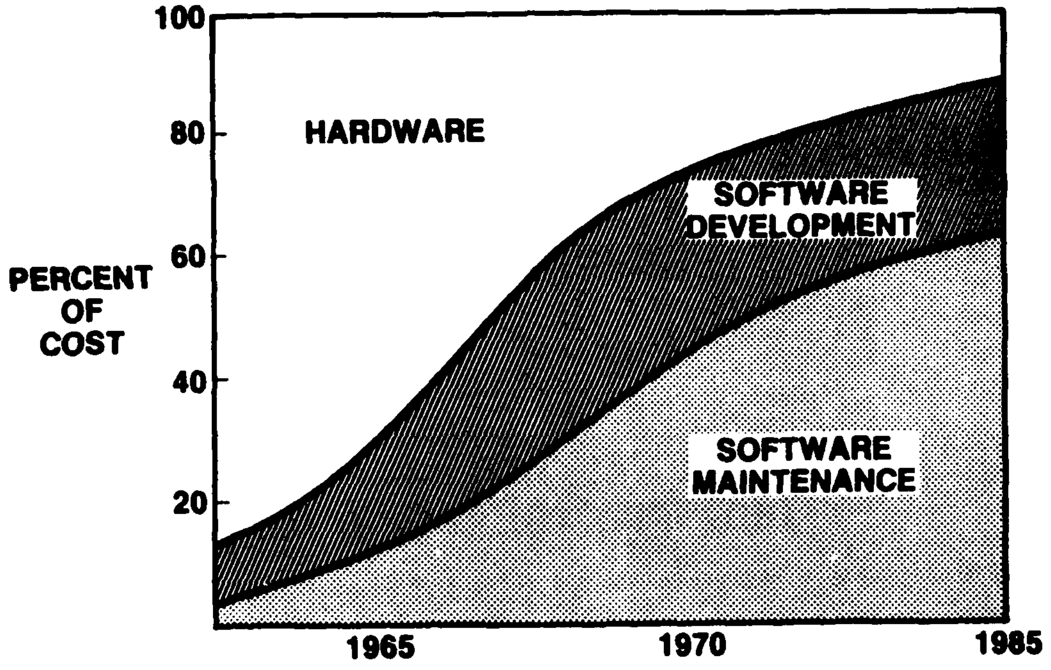


Fig. 2-3 Life Cycle Cost Trends

The cost of DOD software is immense as can be seen from the chart in Figure 2-4. According to this Electronics Industries Association study [5], by the year 1990 the cost of software alone will be approximately $25.6 billion. To put that figure in perspective, the total price tag for the B-1B Bomber fleet was around $20 billion in 1981 dollars. That included 100 aircraft, the initial spares, the weapon system crew and maintenance trainers and the initial logistics support. For the amount of money being spent on software today, the DOD can almost double the new strategic bomber fleet.

In contrast, the amount of money being spent on computer hardware has barely increased. When one considers that computer hardware is more powerful today that ever before, the cost for comparable performance has actually decreased immensely. Many factors have contributed to the decreasing cost of computer hardware but certainly automation has been a major contributor. Unfortunately it still requires a person to program a computer and programmers and other software specialists are expensive.

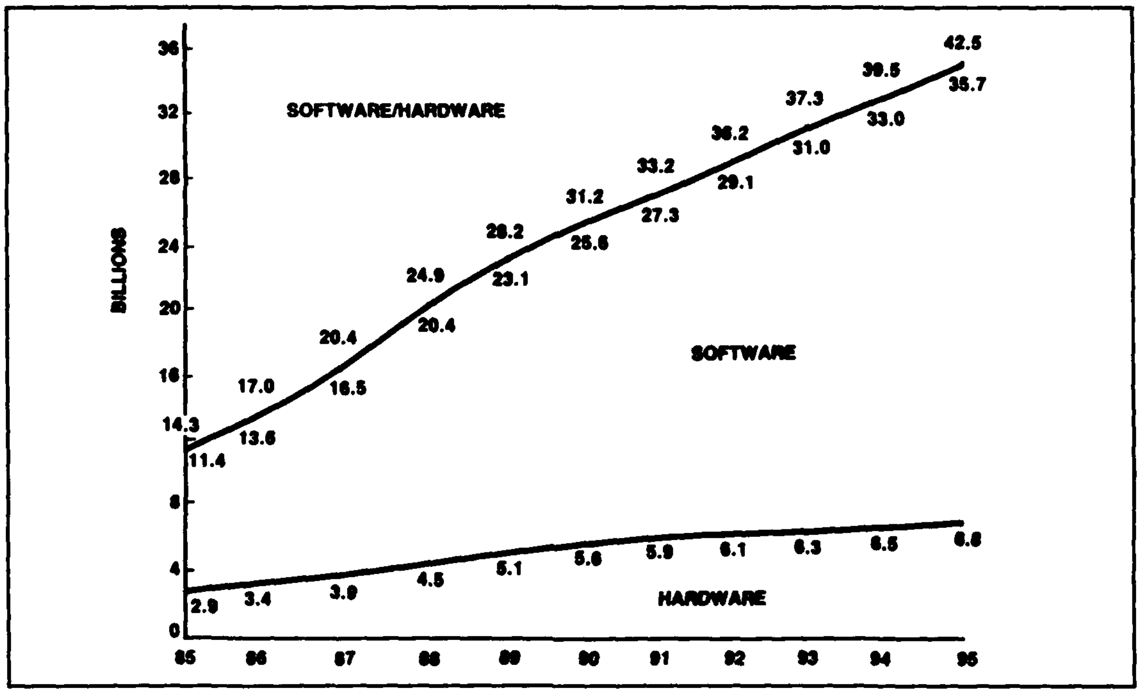


Fig. 2-4 DoD Embedded Computer Market

2.5 EVOLUTION OF DIGITAL SYSTEMS

If one were to make a comparison of Korean War vintage aircraft with modern day aircraft, one realizes that both aircraft contain the same types of functional systems (Figure 2-5). A Korean War vintage aircraft contained flight controls, navigational gear, radar, etc. A modern day aircraft has the same types of systems except it now has a flight control system, an inertial navigation system or INS, a radar, etc. Let's examine some of these systems in greater detail.

A 1952 F-86 Sabre, for example, had a stick which was physically attached to mechanical linkages which in turn were attached to the various hydraulic actuators and control surfaces such as elevators. When a pilot moved the stick, there would be an accompanying movement of the control surfaces because of the mechanical connection. The amount of force required to move the stick depended on how fast the aircraft was flying and its attitude. Contrast this with the latest version of the F-16 where the mechanical linkages have been replaced with electrical wires and motors. Movement of the stick creates an electrical signal which travels down the wires and activates the motors to physically move a control surface. In addition, there may be two or more wires to provide redundancy and a means to artificially give the pilot "a feel" for moving the control surfaces since there is no feedback from mechanical linkages. All of this is done with modern computers. The signals traveling down the wire are digital in nature and the redundancy checks and the artificial "feel" are all controlled by computers. Furthermore, the computer gives the flight control system the

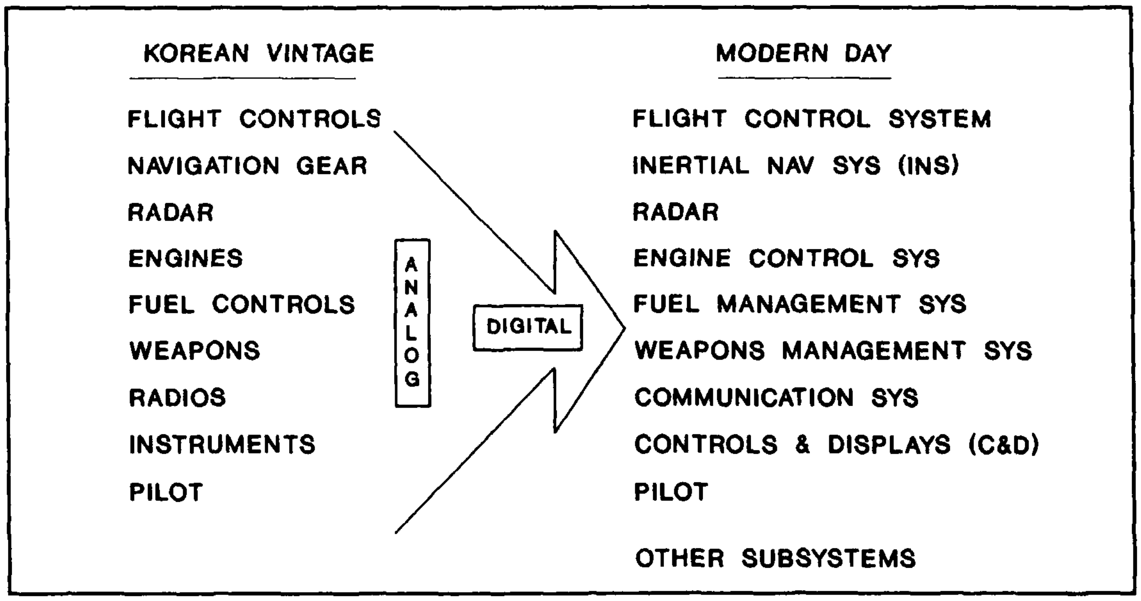


Fig. 2-5 Evolution of Fighter Aircraft

ability to "fly" the aircraft in ways not possible if the pilot were the controlling element. The X-29, the forward swept-wing experimental aircraft, would be virtually impossible to fly without computers. The movement of the stick allows the pilot to indicate where he wants the aircraft to go, but the computer actually flies the airplane.

An F-86, for example, carried navigational gear on board to allow the pilot to find his way to his destination. This consisted of a magnetic compass, an altimeter for altitude indication and perhaps some kind of radio direction finding equipment. With this navigational gear the pilot was able to navigate using dead reckoning techniques. Today we have an Inertial Navigation System (INS) comprised of gyroscopes, accelerometers, and computers to perform the same function. The INS is aligned before takeoff and it allows the pilot to accurately navigate from one point to another. This is only possible because of the computer. Using a known model of the error sources within the INS, the computer uses a mathematical technique known as Kalman Filtering to keep track of the aircraft's precise position over time.

The Korean War vintage cockpit, packed full of instruments, has been replaced with a cockpit containing just a few instruments and controls and display screens. Any information required by the pilot is simply displayed on the screens at the push of a button; anything from attitude indications to the status of weapons. They are all under the control of computers. More importantly, there are now new subsystems that would not be possible without digital systems: diagnostic systems that can display the health of all the major subsystems and "expert systems" that provide the pilot with information on the various options available during a particular mission.

In summary, one can say that computers and software have introduced a whole new dimension to our weapon systems; improved system performance; become an aid to the decision making process; expanded the capabilities of the human operator and in many cases replaced the human operator. In short, they have dramatically enlarged the performance envelope of weapon systems far beyond what was possible less than 30 years ago.

Without modern computers and the associated software, modern weapon systems would not exist. Weapon systems have evolved from systems where computers played a very minor role to systems where their very existence depends entirely on computers.

2.6 WEAPON SYSTEM SOFTWARE

The discussions so far have centered around the computers and the software that are embedded in a weapon system and are an integral part of that system. There is, however, a whole host of software associated with every weapon system (Figure

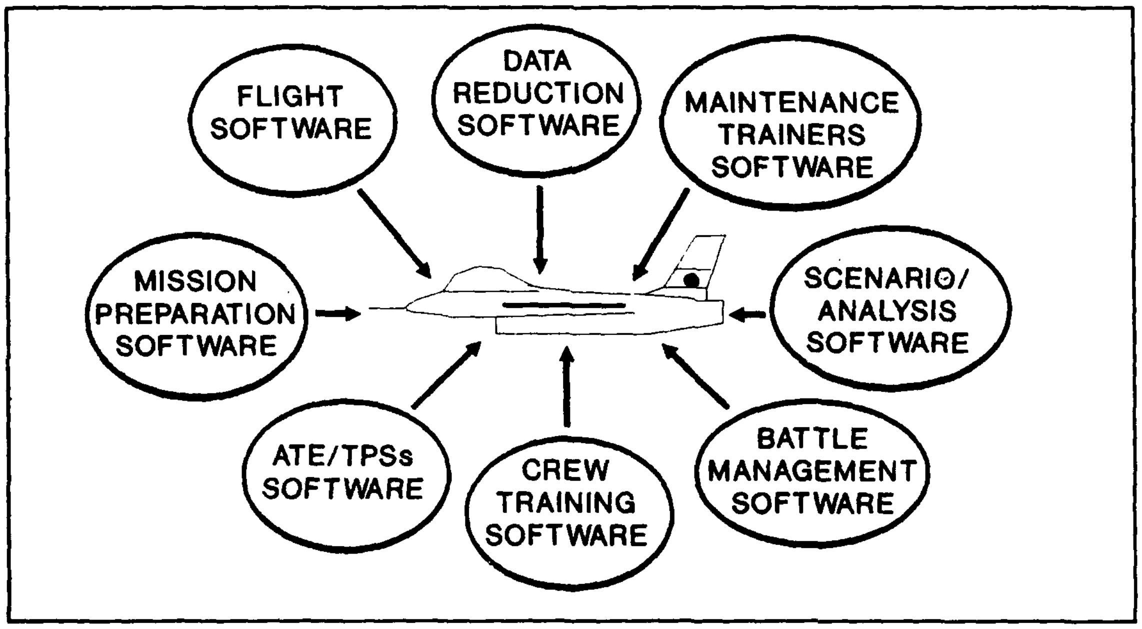

FLIGHT SOFTWARE
DATA REDUCTION SOFTWARE
MAINTENANCE TRAINERS SOFTWARE
MISSION PREPARATION SOFTWARE
SCENARIO/ ANALYSIS SOFTWARE
ATE/TPSs SOFTWARE
CREW TRAINING SOFTWARE
BATTLE MANAGEMENT SOFTWARE

Fig. 2-6 Weapon System Software

2-6) that is not embedded in the system but is absolutely essential nevertheless.

The software that has been discussed so far is referred to as flight software but a wealth of other software is required to support a weapon system. During ground and flight testing, extensive data reduction computers and software are required in order to aid in the analysis of literally millions of data points generated as part of a major test program.

In order to train the various maintenance crews, various subsystem trainers must be developed many of which require literally hundreds of thousands of lines of code and numerous computers, both large and small. Two examples are avionics maintenance trainers and weapons load trainers. Operational analysis personnel require very large scenario software to perform tactics and war planning. Extensive and complex battle management software may be required to develop mission planning. Operational crews usually train on large software intensive weapon system trainers whose functions are dependent on millions of lines of software instructions. For logistics support of electronic equipment, automatic test equipment (ATE) and its associated software must be developed along with hundreds of software packages called Test Program Sets (TPSs) which allow technicians to isolate and repair failed electronic components. Lastly, mission preparation software may also be required for use by operational crews in planning and carrying out their missions.

2.7 CURRENT STATE OF AFFAIRS

Examination of the current state of affairs with military weapon systems reveals some very unpleasant facts:

(a) Most new systems are extremely complex. This is due to a combination of several factors:

- extremely demanding requirements, which tend to drive designers towards complex solutions;

- tight schedules and even tighter budgets, which tend to negate elegant and simpler solutions;

- and, unfortunately, too many contractors who are not fully skilled in software engineering techniques tend to populate the "lowest bidder" category. These contractors seem to thrive under our current procurement laws and regulations.

(b) Digital systems are now the heart and soul of all new weapon systems. The flexibility offered by digital systems cannot be remotely approached by analog systems. This trend will continue for the foreseeable future.

(c) Most systems are delivered late, have cost overruns, rarely meet performance requirements upon initial delivery and are often ridiculously expensive to maintain.

It would be very unfair to blame all of these unpleasant facts just on digital systems and software, but it is generally recognized that software is a major contributor, and often the only contributor, to these problems. Software has become the Achilles heel of weapon systems. Not only is it in the critical path of the system development process but system performance is dependent on the system software.

2.8 HISTORICAL CONTRIBUTORS

One of the major contributors to the problems associated with software development has been loose and very often nonexistent management oversight. Since most program managers know little or nothing about software, they concentrate their efforts on hardware or system issues and often leave software management to managers who are not always part of the mainstream decision making process. They get involved only when software starts affecting the overall schedule and by then it is too late. This hands-off policy is changing but not fast enough.

Another problem is that some software managers lack relevant experience. But this is a universal problem with no quick solutions in sight. Experienced software managers within the government are a scarce commodity. Industry seems to lure the good ones away and those that remain and rise to management positions are not necessarily the most experienced.

A major problem that has plagued the DOD in the past has been the uneven application of standards and, in some cases, the lack of standards. As for the former, too often a contract simply calls out all the applicable standards without regard to the fact that many of them are contradictory or even unnecessary. All standards and regulations should be tailored for each program. It wasn't until recently, that standardized programming standards were applied across all the services.

Throughout the 1960s and 1970s the total number of programming languages used for military systems numbered in the hundreds and none of them were compatible with each other. Most systems used their own language and their own computer so that transportability across systems was nonexistent. To a great extent many systems still suffer from this problem.

Perhaps the most significant contributor to the software problem has been the almost endemic lack of a disciplined engineering approach to software development. The better developers have the necessary discipline to do a good job but even they encounter problems. Unfortunately, many software developers only pay lip service to modern software development methodologies such as top-down structured design, quality assurance, configuration management, and completion of prerequisite tasks prior to proceeding to the next step. They may write convincing proposals but they can't always deliver. On the government side, the problem is compounded when the contractor is not forced to follow a disciplined engineering approach. Too often, engineering discipline is sacrificed to that holiest of sacred cows -- schedule and cost.

Somewhat related is the fact that competent software developers are not sufficient in numbers to satisfy the demands of both the military and the civilian market place. There are more software projects than there are competent software developers, so the less skilled fill the vacuum.

Lastly one must remember that the software that is developed for weapon systems is the most difficult and most challenging type of software. Some of the reasons for this difficulty are the following:

(a) Most weapon systems have real-time requirements which add an additional level of complexity. The software has to respond almost instantaneously and correctly, in spite of noise and other types of interference that can seriously degrade a system.

(b) Most weapon systems have a requirement for fault-free operation or some level of fault tolerance. This adds additional overhead to the software since more checks and redundant capabilities have to be added. These requirements run counter to the requirements for speed, simplicity, and real-time response.

(c) Because of the complexity, many software developments stretch over periods of three to five years. During this time there is usually significant personnel turnover, especially in the government. This results in loss of continuity and provides ample opportunity for new "shakers and movers" to express their leadership and managerial "styles". It also provides plenty of time for Congress to cut, slice, batter and reap havoc with the budget. Although this is also true for hardware, it has a more severe impact on software development.

(d) Because of the long development period and because the threat is always evolving, the designers are shooting at a moving target since requirements are usually not nailed down until late in the development cycle.

(e) No human endeavor is entirely free of politics. The program office is no exception. A program manager must not only deal with management and technical problems but also learn to navigate the more dangerous waters of internal and external politics.

(f) Compounding the problem even further is the sad reality that computer resources technology is rapidly changing. What is state-of-the-art at the beginning of a major weapon system development, is often obsolete by the time the system is fielded. A program manager must be able to properly balance the risk associated with using technology that is at the cutting edge, but which is not yet fully proven, with the risks associated with using more proven but less capable technology.

2.9 PROGRAM MANAGEMENT GUIDANCE

Program managers, as well as software personnel, need to be educated and trained. The Program Managers Course at the Defense Systems Management College is an example of this type of training. But this education and training must be made available to all program office personnel. Even if one is not directly involved with software, that person should still have an appreciation for the difficulties involved since all future hardware development will undoubtedly be impacted by software. Program managers must allow their software personnel to attend courses and seminars so that they can better learn the process. This training is especially valuable for junior and middle level software managers and engineers, many of whom have little or no formal training in software engineering. Program managers must never use the excuse that "we are too busy to let them go now" because they will always be too busy. Make the time and let program personnel attend seminars and training classes.

All program personnel must lose their fear of software. There is nothing magical about computers and software as long as time is taken to learn at least some of the basics. Software and computer illiteracy can no longer be tolerated in a program office.

Software and hardware standards should be intelligently applied. They should be scrutinized and tailored to a specific program. Most standards are written to cover the entire waterfront and particular programs only deal with a portion of that waterfront. Failure to do so will create confusion and will eventually impact those sacred cows -- schedule and costs.

From day one, program managers must pay attention to software and ensure that program personnel are doing the job of enforcing the developer to follow a disciplined process. The chief software person must be made visible by having to report to the program manager on a weekly basis and on a daily basis during critical periods. This person should be totally aware of all of the developer's major activities and have the facts readily available.

Most of all, the program manager must make absolutely sure that program personnel actually read and critically evaluate all software documents submitted by the developer. If they are rubber-stamping documents, then they should be replaced. If necessary, reviews such as Preliminary Design Reviews (PDRs) and Critical Design Reviews (CDRs) should be delayed until the proper documents have been thoroughly reviewed. A developer should not be allowed to slip through a gate until it has satisfied all the requirements for going through that gate; or until everyone is fully aware of the risks involved by proceeding.

As will be seen in a later section of this guide, early and thorough planning is the only way that the software problems can be minimized. This planning starts during the concept exploration phase and continues to some degree until the system is no longer used.

There are no magical solutions! Good software development requires extensive planning and thorough vigilance. There are no short cuts or cookbook solutions!

2.10 REFERENCES

1. Seidman, Arthur H. and Flores Ivan, Ed., "The Handbook of Computers and Computing". New York: Van Nostrand Reinhold Company Inc., 1984.

2. Fairley, Richard E.,"Software Engineering Concepts", Tyngsboro, Mass.: McGraw Hill Book Co., 1985.

3. Grove, H. Mark, "DoD Policy for Acquisition of Embedded Computer Resources", Concepts, The Journal of Defense Systems Acquisition Management, Autumn 1982, Volume 5, Number 4.

4. Boehm, Barry, "Software Engineering", IEEE Transactions on Computers, Vol. C-25, No. 12, December 1976.

5. Electronics Industries Association,"The Military Market: Perspectives on Future Opportunities", Sponsored by the Requirements Committee, Government Division, Nov 1985.

CHAPTER 3
SOFTWARE ACQUISITION POLICY

3.1 INTRODUCTION

This chapter summarizes the DOD's policies governing the acquisition of mission critical computer resources (MCCR). In dealing with policy, it may be useful to understand the history behind its implementation. Towards that end, this chapter will provide a historical perspective of the various laws, regulations, and initiatives that relate to MCCR.

As indicated in the previous chapter, computers and software have become an extremely vital component of a weapon system. In a span of only twenty five to thirty years, the dependence on software and cost of software has grown tremendously. But with this growth, there has been an accompanying rise in the technical and management problems across all of the services.

3.2 BROOKS BILL

Prior to 30 October 1965, there was no form of standardization or control over the procurement of automatic data processing (ADP) equipment within the federal government. On that date, however, Public Law 89-306 (otherwise known as the Brooks Bill) was signed by President Johnson. This bill was intended to promote competition and insure stability in the procurement of ADP resources. By 1976, 36% of the systems were procured in a fully competitive manner and, according to the General Services Administration (GSA), over $681 million in cost avoidance has been achieved in 302 competitive ADP contracts [1].

Traditionally, computers and software have been viewed by top level management as tools for improving efficiency and conserving resources. Although this is true, computer resources need to be treated in the same manner as other acquisitions, not as mere tools. Although primarily directed toward the procurement of ADP equipment, the Brooks Bill forces federal agencies to analyze their ADP requirements, like they would for other systems, and compete for the most economic and efficient system. In an effort to achieve this goal, the Brooks Bill assigned responsibilities as follows: the GSA was given the authority for procuring ADP resources required by federal agencies; the Office of Management and Budget (OMB) was to provide policy guidance and overall leadership (i.e., they were to act as mediator in resolving any user and GSA disputes); and the National Bureau of Standards (NBS) was to develop ADP standards.

The Brooks Bill, however, does not permit the GSA to interfere with an agency's (user) determination of its ADP requirements. The user determines its requirements for ADP equipment and the potential method of procurement. The method of procurement is then approved by the GSA and any disputes resolved by the OMB.

The DOD considers MCCR exempt from the provisions of the Brooks Bill because MCCR is not specifically addressed in the ADP definition. Therefore, the DOD has continued to procure MCCR as part of the weapon system using major system acquisition guidelines. Additional legislation in the form of the Warner-Nunn Amendment and various DOD Directives and Instructions have further approved this interpretation.

3.3 DOD DIRECTIVE 5000.29

DOD Directive 5000.29, "Management of Computer Resources in Major Defense Systems" was published on 26 April 1976. Its purpose was to establish a DOD policy for the management and control of computer resources during the life cycle of major weapon systems.

The directive was the first major step undertaken by the DOD to address the growing software problem. It represented the department's first formal recognition that software is critical to weapon systems and should be managed as a configuration item in the same manner as hardware. As such, software requirements should be validated and risk analyses performed prior to a Milestone II decision in order to insure that the software requirements reflect the operational requirements. All of the management tools used in the development of hardware should be applied to software (e.g., configuration management, baseline\milestone management, and life cycle support planning). The directive also stressed the need for a standardized HOL.

In an effort to help carry out the intent of this directive, a Management Steering Committee for Embedded Computer Resources was also established in 1976 and its charter was contained in the directive. The purpose of the steering committee is to increase the visibility and improve the management of computer resources within the DOD, to formulate a coordinated technological base program for software, and to integrate computer resource policy into the normal process of major system acquisitions.

The steering committee is composed of two boards: the Executive Board and the Management Advisory Board. The Executive Board is responsible for the development of policy necessary for the acquisition and management of computer resources in major defense systems. It consists of a representative from the Assistant Secretary of Defense (Installations and Logistics), who is the chairman, the Deputy Director Research & Engineering; the Director, Telecommunications and Command and Control Systems; the Assistant Secretary of Defense (Comptroller); and the Assistant Secretary of Defense (Intelligence).

The Management Advisory Board is responsible for coordinating technology efforts among the DOD components, for conducting policy impact assessment for the Executive Board relating to computer resources, and for reviewing computer resource technology programs for policy consistency. It consists of representatives from the Navy, Army, Air Force, Office of Joint Chiefs of Staff, Defense Communications Agency, National Security Agency, Defense Advanced Research Projects Agency, and Deputy Director (T&E).

3.4 WARNER-NUNN AMENDMENT

The Warner-Nunn Amendment (Section 908 of Public Law 97-86, the DOD Authorization Act, 1982) was implemented in order to broaden the range of embedded computer resources excluded from the provisions of the Brooks Bill. It was intended to provide the DOD with more control over the acquisition of computer resources that are an integral part of weapon systems.

The Warner-Nunn Amendment defined those computer resources which are exempt from the Brooks Bill. It defined MCCR as those computer resources that perform the following functions, operation, or use [2]:

(a) Involves intelligence activities;

(b) Involves cryptoanalytic activities related to national security;

(c) Involves the command and control of military forces;

(d) Involves *equipment that is an integral part of a* weapon system;

(e) Is critical to the direct fulfillment of military or intelligence missions.

The essential test as to whether the acquisition of computer resources is covered by the Warner-Nunn Amendment or the Brooks Bill is the intended use of the equipment and services, and not their commercial market place availability. Interpretation of these policies must not be lightly rationalized and used as an excuse to depart from sound business and management practices. Where there is doubt as to the applicability, case-by-case determinations shall be made by the Under Secretary of Defense (Research and Engineering)[1], in coordination with the Assistant Secretary of Defense (Comptroller) [3].

[1] Since the time the Warner-Nunn Amendment was implemented, the Under Secretary of Defense for Acquisition was given the responsibility for military software within the DOD.

3.5 MCCR STANDARDIZATION

The explosion in the number of weapon system computer applications that occurred during the late 1960's and the 1970's, resulted in a comparable explosion in the number and types of computers and programming languages. By the mid-1970's, there were literally hundreds of different computer programming languages being used to generate military systems software. Along with these languages, there was an almost equal number of different computers in use. The result was that engineers, technicians, and computer programmers who were supporting a particular weapon system could not support a different weapon system without costly retraining and delays. Rarely would two different weapon systems use the same computer or the same programming language. This led to needless duplication of effort and inefficient use of human resources. Since each system had its own computer and programming language, each system was unique and required unique resources.

In order to minimize weapon system software support costs and to promote interoperability between the various systems, the DOD focused on three areas: higher order languages (HOLs), the software development process, and computer hardware. The specific policy and guidance for each area are the following:

<u>HOLs</u>

DOD Directive 3405.1	- Computer Programming Language Policy
DOD Directive 3405.2	- Use of Ada in Weapon Systems

<u>Software Development</u>

DOD-STD-2167A	- Defense System Software Development
DOD-STD-2168	- Defense System Software Quality Program.

<u>Computer Hardware</u>

MIL-STD-1750A	- Airborne Computer Instruction Set Architecture.

HOLs and computer hardware are discussed in Chapter 4. The software development process described in DOD-STD-2167A and DOD-STD-2168 are addressed in Chapter 5.

3.6 DOD DIRECTIVE 3405.2

Standardization of HOLs has been an issue within the DOD since the early 1970's. In 1976 DOD Instruction 5000.31, "Interim List of DOD Approved Higher Order Languages (HOL)", was issued as an interim measure to limit the number of DOD approved HOLs to six:

<u>DOD</u>	FORTRAN COBOL
<u>Army</u>	TACPOL
<u>Navy</u>	CMS-2 (CMS-2M & CMS-2Y)
<u>Air Force</u>	JOVIAL (Version J3 & J73)

A few years later that instruction was amended to include ATLAS as an approved HOL for automatic test equipment. Concurrently, the DOD initiated a fully competitive program to develop a common, preferred, single HOL for DOD software development programs. The outcome of these efforts was the Ada programming language (see Chapter 4). On 30 March 1987, DOD Directive 3405.2, "Use of Ada in Weapon Systems", was published to establish DOD policy for using Ada as the <u>single, common, HOL</u> in the development of computers integral to weapon systems. Computers are defined as being integral to a weapon systems if they are:

(a) Physically a part of, dedicated to, or essential to the real time performance of the mission;

(b) Used for specialized training, diagnostic test and maintenance, simulation, or calibration;

(c) Used for R & D of weapon systems.

The directive applies to all new weapon systems entering into development (prior to Full Scale Development (FSD)) and to major upgrades (greater than 1/3 modification) to existing systems. There are three exceptions, however, to using Ada:

(a) If a programming language other than Ada was already in use during the FSD phase of a weapon system on the effective date of the directive, then that language may continue to be used through deployment and software support unless the system is undergoing a major software upgrade.

(b) Ada is preferred, but not required, as a test language to be used solely for hardware unit under test equipment.

(c) Ada is preferred, but not required, for commercially available, off-the-shelf software that will not be modified by the DOD.

Except for the conditions stated above, a waiver is required. Authority for issuing waivers is delegated to each DOD component only on a specific system or subsystem basis. For each proposed waiver, a full justification will be prepared and will include a developmental risk analysis; a technical performance, cost, and schedule impact analysis; and a life cycle cost analysis.

3.7 DOD DIRECTIVE 3405.1

The overall policy for DOD computer languages, DOD Directive 3405.1, "Computer Programming Language Policy", was published 2 April 1987, three days after the Ada directive. It superseded DOD Instruction 5000.31 and revised the list of approved DOD HOLs to be used for the development and support of all computer resources managed under DOD Directive 5000.29 and DOD Directive 7920.1 (See Section 3.13). The approved list of the DOD programming languages are: Ada, C/ATLAS, COBOL, CMS-2M, CMS-2Y, FORTRAN, JOVIAL (J73), MINIMAL BASIC, PASCAL, and SPL/1.

This directive stresses standardization of HOLs, and establishes Ada as the *single, common preferred* language within the DOD. When Ada is not used, only the other approved standard languages listed shall be used. This directive serves to limit the number of HOLs used in the DOD and facilitates the transition to Ada. The order of preference is based upon life cycle cost and impact as follows:

(a) Off-the-shelf applications packages and advanced software technology;

(b) Ada-based software and tools;

(c) Approved standard HOLs.

3.8 THE Ada PROGRAMMING LANGUAGE

After the successful development of the first HOL compiler, many new HOLs were quickly introduced. By the mid-1970's, the proliferation of languages within DOD had resulted in an unwieldy logistics problem. Hundreds of different languages were being used and each had its own unique support requirements. In 1975, Malcolm Currie, then Under Secretary of Defense for Research and Engineering (USDRE), suggested that the DOD consider using just one software language. In 1976, an interim policy was issued requiring the use of "approved" higher order languages listed in DOD Instruction 5000.31. At about the same time a committee called the Higher Order Language Working Group was formed to review existing HOLs to determine candidates for a single DOD language. Their charter was to look for HOLs specifically geared for weapon system acquisition. One of their goals was to find a language that supported both real-time processing and large scale programs. The HOL was also supposed to support modern programming techniques and practices such as top-down and structured design.

Their January 1977 evaluation concluded that no existing language met all the DOD requirements but that some, such as Pascal, Algol, and PL/1 could form a good basis for designing a "new" language. In July 1977, as a result of an extensive evaluation process, a new set of language requirements was established. The government initiated four contracts for the design of this new DOD language. After exhaustive design evaluations, a final selection was made for one contractor to develop the new language which became known as Ada. The language was named after Lady Augusta Ada Byron, Countess of Lovelace, who is credited with being the first programmer. She was a well educated mathematician who suggested and wrote the first programs for Charles Babbage's "Analytic Engine", a predecessor of the modern computer. The winning contractor was the French based Honeywell-Bull Corporation, and the design team was led by Jean Ichbiah. Their product, the Ada Programming Language Specification, MIL-STD-1815, was officially published on 10 December 1980. In June 1983, it became an American National Standards Institute (ANSI) standard and was officially published as ANSI/MIL-STD-1815A on 22 January 1983. In March 1987 it became an International Standards Organization (ISO) standard. The European software community has been quick to adopt the Ada Programming Language.

In June 1983 when Dr. DeLauer, then the USDRE, directed that Ada be used on all new major programs, he had been led to believe by the DOD software community that the necessary technical support (i.e., compilers) for Ada would be ready. Unfortunately, this was not the case and this lack of supporting tools is probably responsible for a majority of the bad initial publicity Ada may have received. The fact remains, however, that Ada is a good language for use on MCCR as well as for general purpose applications. What was initially missing were the support resources required to develop Ada software; the same resources that are required to develop software in any HOL. The need for support was well recognized and initial compilers and tool sets to aid in the development of Ada software are now becoming available.

The major concept behind the use of Ada is the enforcement of modern software engineering principles. It is the real strength behind the Ada programming language.

3.9 SOFTWARE ENGINEERING AND TECHNOLOGY

A great deal of effort has been applied to the area of software engineering, especially the application of systems engineering to the software development process. Numerous organizations have been created and/or tasked with evaluating methods for improving software quality and reliability; for reducing development and support costs; and for controlling the management of software development. The main programs that have been initiated are:

(a) Very High Speed Integrated Circuits (VHSIC) - The VHSIC Program Office was created in 1980 by the DOD in order to alleviate deficiencies in military integrated circuits (ICs). Unlike the early days of IC technology, the DOD was no longer the driving force behind technological innovation nor the largest user of IC chips. The commercial market place was now dictating IC developments by virtue of the fact that it was by far the largest consumer of IC products. This meant that military applications had become a specialty and a highly customized business. The commercial market place had no need for extremely fast, highly specialized, low-volume IC products. So the DOD took the initiative, through the VHSIC program, to accelerate the development of this technology.

Another problem plaguing military electronics has been the unusually long time it takes to incorporate a new product into a weapon system once the product is introduced into the commercial market place. It is not unusual for a product to appear in a weapon system up to five years after its commercial appearance. In order to speed up this process, the VHSIC Program has supported the development and insertion of VHSIC chips into military systems. This gives developers and acquisition managers a military qualified microelectronic technology that is on a par with commercially available technology. At least twenty seven major system insertion efforts are presently being undertaken in which the use of VHSIC technology provides the potential for improved system performance [4]. As great advances are being made in computer hardware, the challenge to the software community is to capitalize on this wealth of hardware technology. Software needs to be developed to fully use the capability of this new hardware.

(b) Software Technology for Adaptable Reliable Systems (STARS) - The STARS Program Office was established in 1983 by the DOD to investigate ways of reducing software development costs, to increase software systems reliability, to investigate software automation techniques, and to look at applications for reusable software.

(c) Software Engineering Institute (SEI) - The SEI, located at Carnegie-Mellon University, was placed under contract by the Air Force (Electronics System Division, Hanscom AFB) in 1984. They are were tasked with investigating the transition of new software technology, analyzing software development environments, and providing education in the software and system engineering process. The SEI has recommended changes to the Federal Acquisition Regulations on software data rights provisions [5]; developed an educational program on software engineering; established liaisons with a variety of educational institutions in order to disseminate curriculum information and material for both undergraduate and graduate education; and conducted numerous software engineering conferences.

(d) <u>Defense Science Board (DSB)</u> - A DSB Task Force on software was originally convened in 1981 by the Under Secretary of Defense (Acquisition) to review a draft DOD Instruction on standardizing computer hardware. It recommended cancelling any further tasking in this area. The Air Force, however, had already developed a standardized ISA for a sixteen bit airborne computer (MIL-STD-1750A). Later the DSB was tasked with reviewing overall software acquisition, management, and computer resource technology procedures and providing recommendations for rectifying any problems. The Task Force's report published in September 1987, stated that the major problems with military software development are not technical problems, but management problems. They recommended that the DOD re-examine and change the attitudes, policies, and practices regarding software acquisition [5].

3.10 SOFTWARE SUPPORT

The Joint Logistics Commanders Joint Policy Coordination Group on Computer Resources Management established a sub-panel in 1979 to specifically look into post-deployment software support (PDSS) and the procedures required to provide adequate software support during and after transition. PDSS will be discussed in Chapter 7.

3.11 TOP LEVEL SERVICE DIRECTIVES AND GUIDELINES

The following is a list of the guidance documents for the respective services. All of these documents stem from the guidance provided by DOD Directive 5000.29.

<u>Navy</u>

SECNAVINST 5200.32
Management of Embedded Computer Resources in Department of the Navy Systems, 11 June 1979

OPNAVINST 5200.28
Life Cycle Management of Mission Critical Computer Resources for Navy Systems, 25 September 1985

OPNAVINST 5230.21
Instruction on standard embedded computer resources

NAVELEXINST 5200.23
Instruction on general software management

TADSTANDS
Tactical Digital Systems Standards b through d on standard computers, programming languages, and reserve capacities.

Air Force

AFR 800-14
Life Cycle Management of Computer Resources in Systems, 29 September 1986

AFSCP 800-14
Air Force Systems Command Software Quality Indicators: Management Quality Insight, 20 January 1987

AFSCP 800-43
Air Force Systems Command Software Quality Indicators: Management Insight", 31 January 1986

AFSCP 800-45
Software Risk Management (Draft)

AFSCP 800-5
Software Independent Verification and Validation (IV&V) (Draft)

ASDP 800-5
Software Development Capability/Capacity Review, 10 September 1987

Army

DARCOM-R-70-16
Management of Computer Resources in Battlefield Automated Systems, 16 July 1979

Asst Secretary of Army Policy Letter
Standardization of ECR, 1 July 1980

AMC-P 70-13
AMC Software Management Indicators, 31 January 1987

Marines

MCO 5200.23
Management of ECR in the Marine Corps, 19 August 1982

3.12 SOFTWARE DATA RIGHTS

The program office must address the issue of data rights in all of its acquisitions. Before the issues of software data rights are discussed, a few definitions should be understood.

Copyright - Protects the expression of an idea through unauthorized copying or reproduction. Easy and inexpensive to obtain by simply filing an application with the copyright office with a copy for the Library of Congress. No examination of the material is required.

Trade secret - Protects the underlying ideas, concepts, procedures, formula, pattern, device or compilation that derives economic value by not being readily available to others. Must maintain secrecy (information is not public domain).

Patent - Protection by trademark or trade name. Expensive, uncertain, and time consuming (may take two years to obtain). Destroys trade secret protection because patent disclosures are quite complete.

Proprietary - A form of trade secret, copyright, or patent. Provides protection to the contractor.

Restricted rights - Trade secrets developed at the private expense of the contractor.

License agreements - Agreement between a contractor and the government (or another contractor) limiting the use and the copying of data which has been commercially sold (rights to use, disclose, or reproduce).

The Federal copyright laws, patent laws, and state trade secret laws are the main laws providing legal protection of a contractor's computer software. The Copyright Act of 1976 was amended in 1980 to include computer programs. The Office of Federal Procurement Policy (OFPP) was tasked by a 10 April 1987 Executive Order, to develop a firm national policy in favor of commercial rights [6]. In the meantime, FAR Subpart 27.4 and 52.227-14 provide guidance to the program office on software data rights. Specific language in accordance with these clauses of the FAR should be included in the contract, both for the protection of the government and the contractor.

3.13 AUTOMATED INFORMATION SYSTEMS

A program manager requires the use of many tools to successfully manage the program office organization. One such tool is an automated information system (AIS) for configuration control, intra-office communications, and other administrative tasks. Since this chapter primarily deals with computer resources associated with weapon systems, it may be useful to briefly discuss the DOD policy on AIS.

After the DOD had taken steps to provide initial guidance on the management aspects of weapon system computer resources, it published DOD Directive 7920.1, "Life Cycle Management of Automated Information Systems (AIS)" on 17 October 1978 and DOD Instruction 7920.2, "Major Automated Information Systems Approval Process" on 20 October 1978. These two documents established the policy for the procurement of general purpose computers or ADP equipment primarily intended for use in business applications and subject to the provisions of the Brooks Bill. AIS defines procedures for the acquisition of equipment which is designed, built, operated, and maintained for the sole purpose of collecting, recording, processing, storing, retrieving, and displaying information. These systems usually have large data storage requirements and are used for business type applications such as payroll, accounting, and inventory. The function of these two DOD documents is very similar to that of the DOD Directive 5000.29. Some of the similarities include the promotion of life cycle management, visibility, cost effectiveness, standardization of the approval process, and emphasis on requirements validation.

3.14 SUMMARY

Proliferation of software and computer resources has occurred since their introduction in the late nineteen fifties. The Brooks Bill was passed in 1965 to promote competition and to regulate how ADP should be acquired and managed within the government. The emphasis in this chapter, however, has been on the acquisition policy for software exempted from the Brooks Bill.

The DOD provided guidance on the management of software and MCCR in 1976 with DOD Directive 5000.29. The Management Steering Committee for Embedded Computer Resources was established to guide this effort. This was followed by software workshops in 1979 conducted by the Joint Logistics Commanders' Joint Policy Coordination Group on Computer Resource Management. Standardization of the software life cycle has been the main DOD focus.

3.15 REFERENCES

1. Thirty-Eighth Report by the Committee on Government Operations,"Administration of Public Law 89-306, Procurement of ADP Resources by the Federal Government", 1 October 1976.

2. Section 2315 of Title 10, United States Code.

3. Memorandum of Deputy Secretary of Defense, "Acquisition of Automatic Data Processing (ADP) Equipment and Services", 1 February 1982.

4. VHSIC Program Office, "VHSIC Annual Report for 1986", Office of the Under Secretary of Defense for Acquisition, 31 December 1986.

5. Report of the Defense Science Board Task Force, "Military Software", Office of the Under Secretary of Defense for Acquisition, September 1987.

6. "Commercial Rights Will Be Protected", Washington Technology, 17 December 1987.

CHAPTER 4
TECHNICAL FOUNDATIONS

4.1 INTRODUCTION

This chapter addresses the basics of computer hardware and software by describing how a computer works and by defining the concepts of computer programs and languages. These basics are intended to provide the uninitiated with an understanding and appreciation for why a software development project must follow a logical and proven process.

This chapter also provides a brief technical description of the Ada Programming Language and a brief explanation of its technical and management benefits.

4.2 INSIDE THE COMPUTER

In general terms, a computer is a device which receives or "senses" data through input devices, processes that data and provides an output in the form of information or an action. This is illustrated in Figure 4-1. The incoming data can originate from a human operator, external sensors, or computer models used to simulate the external environment.

The incoming data is processed by a computer program which is a set of instructions and data that were previously loaded and stored in the computer. The details of how these instructions and data are generated and stored in the computer will be discussed later in this chapter.

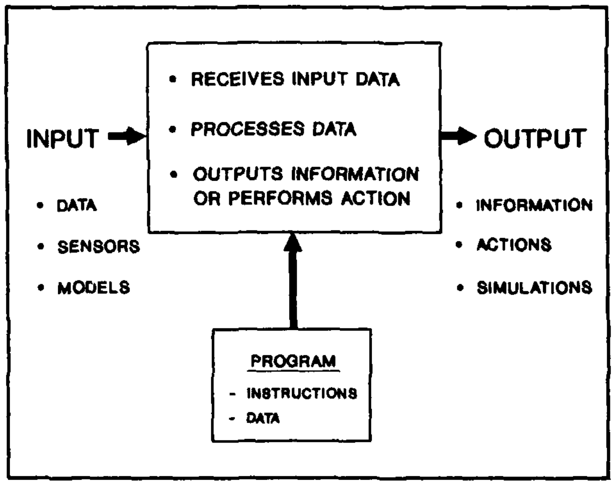


Fig. 4-1 Computer Definition

Once the data has been processed, the computer provides information to a human operator; performs a particular action such as turning on an actuator or updating a data base; or provides processed data to a computer simulation as part of its own input data.

The major components of a computer are the input and output (I/O) section, the central processing unit (CPU) and the memory unit as depicted in Figure 4-2.

4.2.1 Input/Output Section

In order for the computer to be a useful device, it must be able to communicate with people or devices outside of itself. This communication is accomplished through input and output devices. Examples of input devices are terminals, keyboards, and sensors such as navigational instruments, altimeters, fuel level sensors, and temperature sensors. Examples of output devices include printers, displays, actuators, and electro-mechanical devices that are part of a weapon system. In some instances, the two functions are combined as they are on a terminal which includes a display screen and a keyboard assembled into a single unit. In short, it is the computer I/O that provides the interface to the rest of the system.

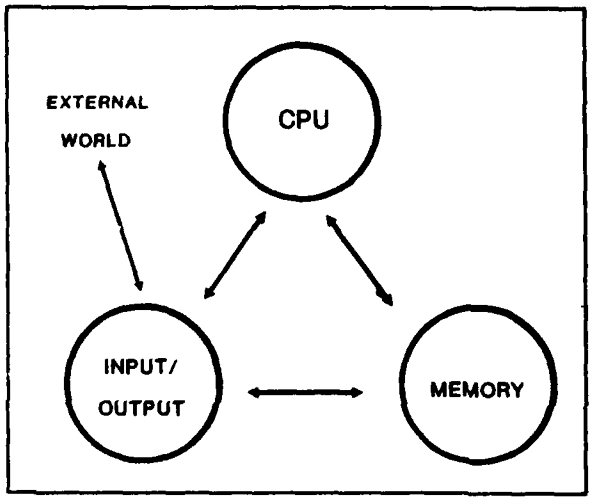


Fig. 4-2 Computer Components

4.2.2 Central Processing Unit

The central processing unit is the brain of the computer. It is in the CPU where the actual processing or computations take place. As already stated, the processing is based on the computer program or set of instructions which have been stored in the computer's memory.

As an example of the type of processing to be performed by a CPU consider an aircraft avionics system which uses an inertial platform and a computer program stored in memory to perform the navigational function. An inertial platform is a device that utilizes gyroscopes to maintain a fixed attitude with respect to some external reference, usually the stars or the earth's north pole, and accelerometers for measuring acceleration. The process of navigation would involve the following: **Step 1:** Begin the process. This will require some housekeeping and initialization to tell the computer the initial starting position and the direction in which the platform is pointing. **Step 2:** Obtain input data from the navigational sensors: i.e.,the accelerometer outputs and the gyroscopic attitude output. **Step 3:** Compute current position and velocity based on the internally stored program. **Step 4:** Output this information to the operator and/or guidance system for further action. This process will be repeated at regular intervals in order to provide a continuous flow of navigational information. In order for the output to be timely, the process needs to occur in real-time.

The CPU is in control during the entire time it is executing these instructions. Step 3 above processed the data received from the platform to produce the necessary information. In this case, the computation involved determining the change in direction relative to the referenced stable platform and calculating the velocity by numerically integrating the acceleration over time. By performing this process in real-time and providing a continuous output, the aircraft's position and velocity will be known at all times.

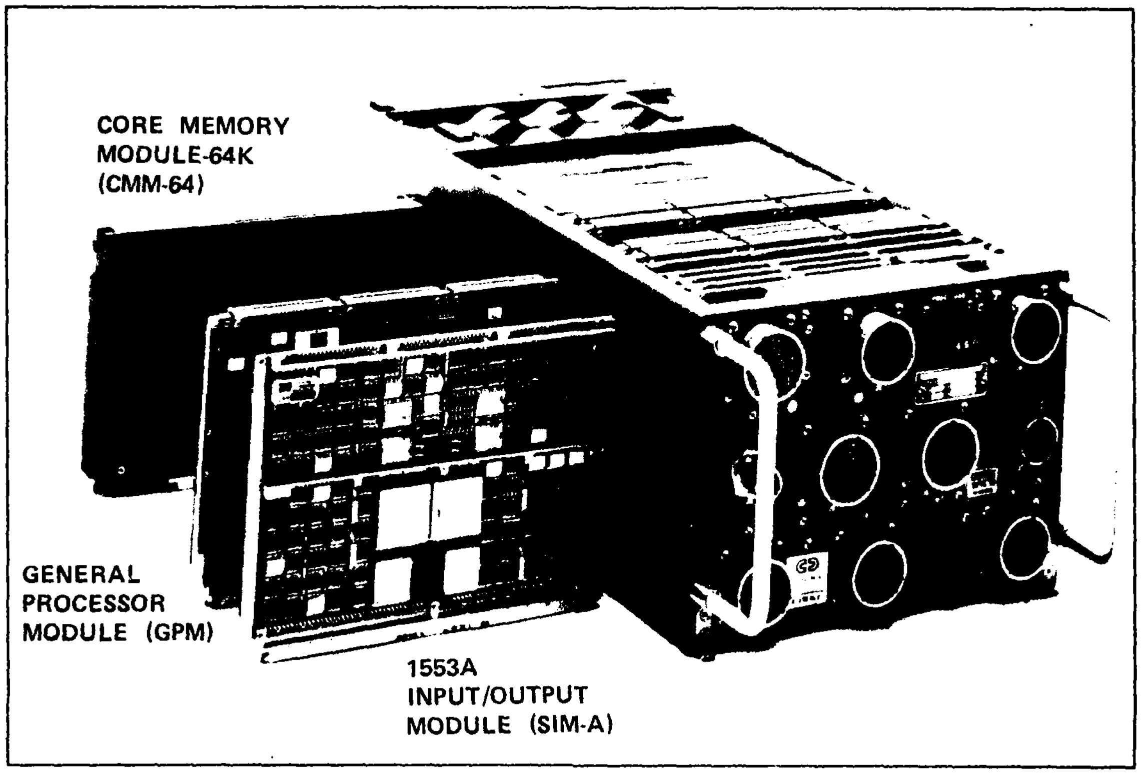


Fig. 4-3 AN/AYK-14 Computer

4.2.3 Memory Unit

The third major component of the computer is the memory unit. One of the important aspects of a computer is the ability to store in its memory the instructions and data required for the computer to perform its functions. By storing different instructions and data, the computer can perform many different tasks within the bounds imposed by the system's design and implementation. In the previous example, a set of instructions stored in memory allowed the computer to perform navigational computations. Using a different set of instructions, the computer could be used to determine the health and status of the hardware components of the entire system. This assumes that the appropriate input data is provided by the various subsystems. The computer has the capability to perform these functions and many more. This ability to store and execute programs provides considerable power. During execution of a program, the computer fetches an instruction out of memory, performs that instruction and then steps to the next instruction. This continues until all the instructions have been performed. Memory is located within the computer, but memory can also be located externally in memory devices such as fixed disks, or tape drives. Internal memory is limited in capacity so external devices, which have more capacity, are used for long term storage of large programs and data.

4.2.4 Computer Hardware

Computer hardware comes in many shapes and sizes. Figure 4-3 shows the Navy standard computer, the AN/AYK-14, which is used in the Navy's F/A-18 aircraft. This computer is about the size of a bread box and its performance is comparable to the original IBM personal computer. There are two of these plus twenty-eight other special purpose processors in the F/A-18.

A much larger system is shown in Figure 4-4. This system is the IBM System/400 computer and it is much more powerful and larger than the AN/AYK-14 computer.

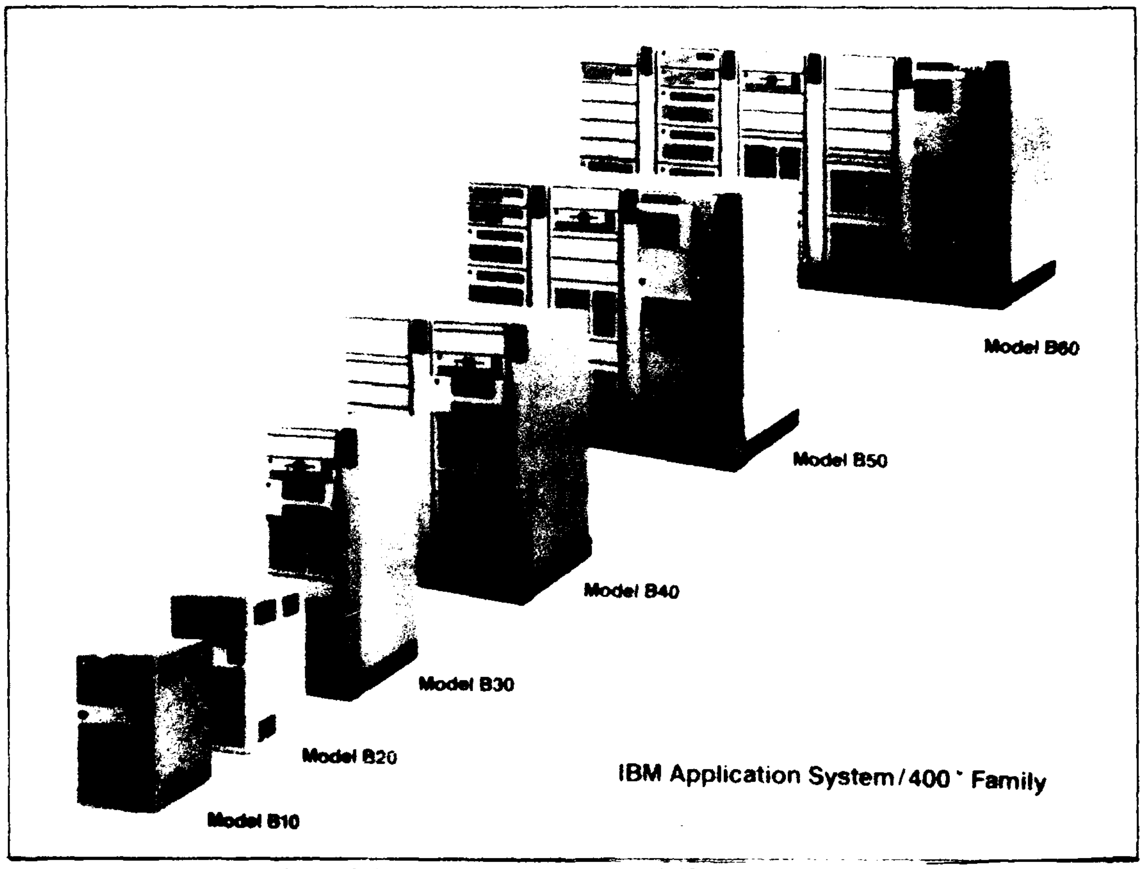


Figure 4-4. IBM Application System/400 Computer Family

4.3 COMPUTER RESOURCES

Now that the basics of a computer have been introduced, it is time to address the bigger picture of what is commonly referred to as computer resources. All too often important aspects of a software project are arbitrarily relegated to positions of insignificance because their importance is not understood and, therefore, receive little or no program management attention until it is too late. To preclude this from happening, program managers must be familiar with all the components that make up and support a computer system. Only by fully understanding all the pieces of the puzzle, can they properly manage computer resources. This doesn't mean that they must have detailed knowledge of a computer processor's operation or have the ability to generate software code for their project. But they should have a basic understanding of computer resources and know how these resources fit into the overall weapon system architecture. The components of computer resources are shown in Figure 4-5.

4.3.1 Embedded Computer Hardware

In weapon systems, the program manager needs to be concerned with mission critical computer resources as defined in Chapter 2. Generally, a weapon system is designed with a special purpose computer because of weight, power, application, or other technical considerations. The special purpose computer or processor may take the form of a "blackbox", an assembly of cards, or even a single card which is embedded in the system. This means that the computer is an integral part of the weapon system.

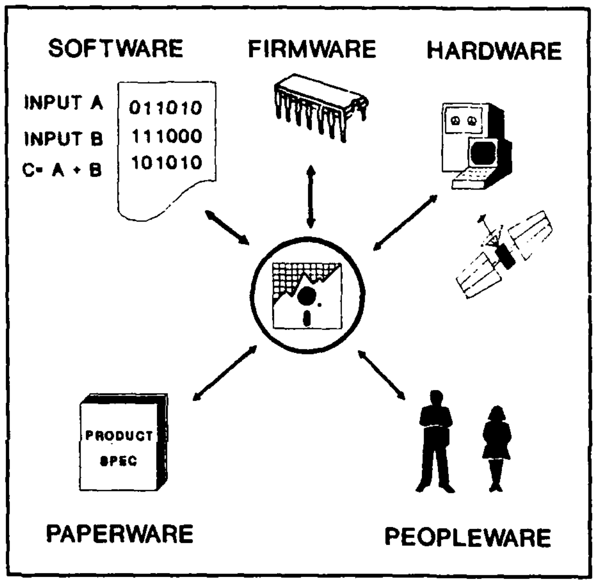


Fig.4-5 Computer Resources

With today's Very High Speed Integrated Circuit (VHSIC) technology, a computer can be built on a single integrated circuit, a piece of silicon not much larger than a 1/4 by 1/4 inch square. To the untrained eye, an embedded computer system may be physically indistinguishable from the rest of the system. An example would be a computer used in the flight control system of an air-to-air missile or the navigational computer in a satellite.

4.3.2 Software

Software is defined by the Federal Acquisition Regulations (FAR) as the set of instructions and data that are executed in a computer. This definition clearly distinguishes data items, such as documentation and specifications that are called out in the contract, from the deliverable software such as an operational flight program. Although some common definitions of software often include all the documentation as well, the DOD definition includes only the executable form of the instructions and data. Software is not something you can touch or feel. It is intangible: is has no mass, no volume, no color, no odor, no physical properties. It can only be represented by a listing or other forms of documentation. Software will be addressed in greater detail later in this chapter.

4.3.3 Firmware

The evolution of computer hardware has also brought about the marriage of hardware and software in a combination called firmware. Firmware is defined as software that has been implemented in hardware using memory devices such as read only memory (ROM), programmable ROM (PROM), erasable PROM (EPROM), and electrically erasable PROM devices (EEPROM). These devices, and other similar devices, allow software to be permanently implemented and not easily changed. In order to change software that has been implemented in firmware, one of two actions must be taken. If the firmware is ROM or PROM, then these memories or ICs must be physically removed from a circuit card and replaced with other ROMs or PROMs that have been programmed with the new software. If the firmware is EPROM, then the ICs must be removed, reprogrammed, and reinstalled. The EEPROM can be altered in circuit but this requires special additional equipment or circuitry. The EPROM is usually altered using an ultra-violet light source and the EEPROM can be altered using electrical means.

Because of the difficulty encountered in changing software that has been implemented in firmware, firmware is used only in applications that:

Require Speed - Many signal processing applications such as electronic warfare systems must receive, analyze, categorize and jam radar signals from hostile threats almost instantaneously. They cannot tolerate the relatively slow processing speeds associated with general purpose computers. In these cases the various algorithms or programming steps are implemented in firmware in order to significantly increase the processing speed.

Require protection from unauthorized tampering or alterations - The software required to run a computer is oftentimes stored in firmware. By using these devices, computer manufacturers preclude programmers from inadvertently changing the resident software, commonly referred to as the operating system software, and possibly causing the computer to fail or to operate improperly.

Require permanent software - Programs that have been implemented in firmware are immediately available as part of the memory and do not have to be loaded when the computer is first powered up. This also provides a form of protection from power failures. A thoroughly tested and stable program is a good candidate for firmware.

Firmware also introduces an additional dimension to software. Because it is software, all the configuration management practices associated with software also apply to firmware. Once the software is implemented in firmware, the ICs are managed as hardware configuration items. In order to provide for future support, a method must exist which traces the specific software version to a unique piece part.

4.3.4 Peopleware

People are also an important part of the system. The program manager tries to satisfy the user's need through a reasonable system design, but it is the user and support personnel who are the ultimate judge of the delivered product's quality. This is why it is important that the program manager involve the user in defining requirements, in evaluating test results, and interpreting system interface requirements. Other chapters will address the importance of involving the user and support personnel in the development process.

4.3.5 Documentation

Because software development is largely an intellectual exercise, documentation is vital for communicating during the software development and support phase. Documentation must be a logical by-product of the development process. As software development tools and computer programming languages become more sophisticated, documentation will become more and more of an automatic by-product of the development process. Until then, however, the program manager must insure that adequate documentation exists to accommodate development and follow-on support. It is important to remember that software is intangible, with no physical properties, and that documentation is the only means available for describing and keeping track of its development progress.

4.3.6 Development/Support Facilities

A computer system consists of hardware, software, firmware, peopleware, and paperware as indicated in Figure 4-5. All these elements, which are brought together in a support facility, must be available during the development and support phases of the weapon system. The support facility is an important aspect of computer resources. It includes not only the physical property, such as the building, host computers, and utilities, but also the supporting software documentation needed for development and support. A software development facility and a software support facility are virtually identical since the same software and hardware tools are required for both. The facility may consist of a host computer, which may be either a large mainframe computer or a minicomputer, along with terminals for the programmers, analysts, testers, librarian, and other personnel (Figure 4-6). The facility may also be a networked system of individual work stations. In order to perform software development and support, several software programs are required. These programs include compilers, linkers, loaders, simulators, editors, and other development and management tools.

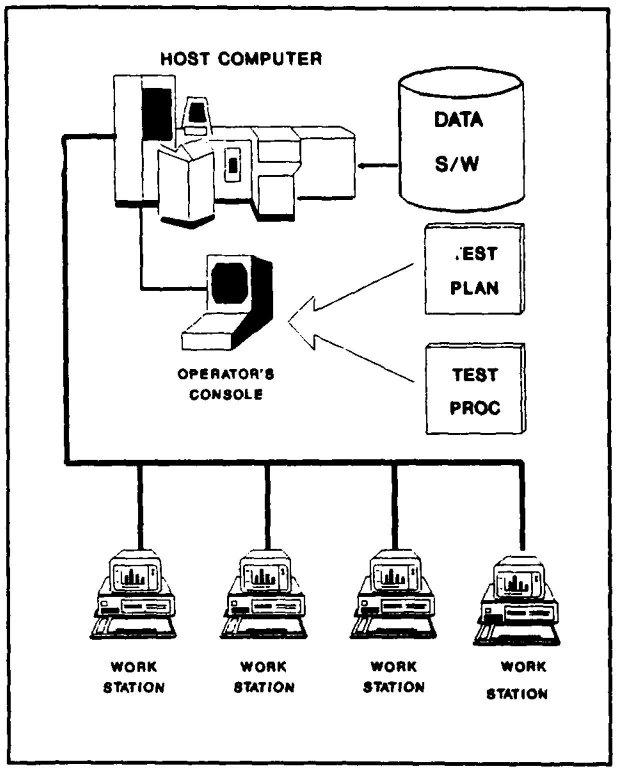


Fig. 4-6 Support Facility

4.4 COMPUTER ARCHITECTURE

The computer can be thought of as a collection of hundreds of thousands of electrical switches. Each of these switches can be in one of two states, on or off. Since the switch has two states, the status of any one switch can be represented by a "0" or a "1", i.e., on = 1 and off =0. The binary numbering system can be used to represent the state of these switches since it too has only two digits, 0 and 1. Instructions and data can therefore be represented by a string of 0's and 1's and by using the rules of Boolean logic, named after the English mathematician and logician George Boole. These switches are interconnected to build modern electronic computers. Modern computers, no matter how large or how small, perform the following basic types of operations or instructions:

a) Arithmetic	(add, subtract, multiply, divide)
b) Logical	(AND, OR, NOT, EXCLUSIVE OR)
c) Transfer control	(branching, loops, subroutines)
d) Data movement	(load, store, move)
e) Input and Output	(in, out)
f) System	(HALT, interrupt)

4.4.1 Bits and Bytes

In the binary numbering system a bit represents one digit, either a "0" or a "1". A word is a string of bits that represent instructions or data; the larger the string the more information it can represent. Any character can be represented by using "coding" techniques. One widely used technique is the American Standard Code for Information Interchange (ASCII) which is used for encoding the alphabet, numbers, and other special characters. There are 128 characters in the ASCII set and 256 characters in the IBM set, another widely used coding technique. As an illustration, part of the ASCII alphabet and number coding scheme is listed below:

CHARACTER	BINARY CODE	DECIMAL EQUIVALENT
0	00110000	48
1	00110001	49
2	00110010	50
3	00110011	60
	.	.
	.	.
A	01000001	65
B	01000010	66
C	01000011	67
D	01000100	68
E	01000101	69
F	01000110	70

Notice that the ASCII standard uses eight bits or digits for the binary representation. This eight bit word length is commonly referred to as a BYTE and was usually the smallest word size in earlier computers, particularly in personal computers (PCs). The eight bit structure of PCs has been replaced by the 16 bit word, with 32 bit structures quickly taking their place. Large mainframes have always used larger word sizes such as 32 or 64 bit word lengths. The advantage of a larger word length is that it contains more information in a single word and can access larger segments of stored data. A computer architecture is designed around specific word sizes since the internal communication between the CPU, memory and I/O is dependent on the number of bits in a word. For example, a computer that has an eight bit architecture (8 bit buss) communicates eight bits at a time (in parallel) while a 16 bit machine communicates 16 bits at a time. This effectively doubles the throughput.

4.4.2 Instruction Set Architecture

The computer architecture and its internal logic structure is designed and implemented by the computer manufacturer to perform a finite and fixed set of instructions. A computer with a minimal set of instructions can perform the same computations as one with a larger set. The difference, however, will appear in the execution time and the sequence of instructions in the software program. Let's assume that a programmer is required to generate a computer program to perform a particular task. A program written for a machine with a large set of instructions will usually require fewer lines of machine instructions than a program written for a machine with a smaller set of instructions. The difference in actual machine instruction sets is dependent on the manufacturer's objectives in design. Computers can be designed and optimized for specific applications. Some computers are designed to perform very rapid mathematical computations; others are designed to manipulate large amounts of data in a very efficient manner; and still others are designed with a very powerful graphics capability. No computer, however, can be built so that it can perform equally well in all applications. There is no industry standard for computer design and each manufacturer is free to design and target their machines for the application of their own choosing. This means that each computer has its own internal and fixed repertoire of instructions. This fixed set of instructions is called the computer's instruction set architecture (ISA) and in order to execute a computer program on a particular machine, that program must be specifically targeted or written for that machine's ISA. In other words, the binary instructions and data that make up a software program are different for computers with different ISAs. Instruction set architectures are the "blueprints" that describes the interface to the set of electronic hardware or circuitry to execute the different types of operations or instructions.

Word size is an important part of the computer architecture. Recall the earlier discussion on the communication of instructions and data within the machine. Part of the basic design is determining the internal signal communication paths. This internal communication is accomplished through the use of an electronic component called a buss. The buss provides parallel signal paths between the CPU, memory, and external devices. A computer will typically have two busses as shown in Figure 4-7. The design architecture will also determine the internal communication within a computer. A computer that has a 16 bit architecture communicates internally 16 bits at a time (in parallel). A computer that has a 32 bit architecture communicates 32 bits at a time. This effectively gives the computer with 32 bits a greater throughput or faster execution capability.

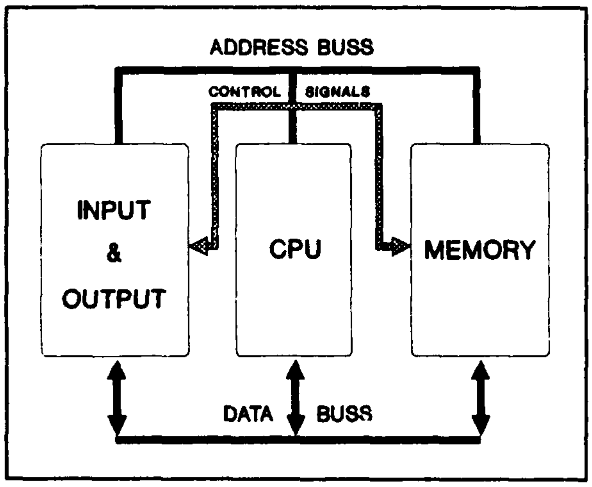


Fig.4-7 Computer Busses

Although there are no official standards for commercial hardware designs and computer architectures, the surge of sales in personal computers has made de-facto standards of some Intel and Motorola computer architectures. Within the DOD, however, there does exist a standard for ISA, namely MIL-STD-1750A. This standard has been used successfully in both Air Force and Navy programs but its application is usually limited to airborne and embedded computer applications. This is because this ISA was initially designed by the Air Force in the 1970s around a 16 bit word size. It was specifically intended for airborne applications and it has a limited memory capacity. Today's rapidly evolving computer technology is quickly making MIL-STD-1750A ISA somewhat obsolete. The advantage of standard ISAs is portability of executable software.

4.5 SOFTWARE LANGUAGES

Software languages are the vocabulary or lexicon used to instruct computers on the functions they will perform. Software languages are categorized into three groups: machine language, assembly language, and higher order language.

4.5.1 Machine Language

Machine language is the most primitive and basic of all the languages and the only language understood by a computer. It is written in binary code and provides the machine the instructions it is to execute. The binary coded words are those words that were designed for the machine's ISA. Programming in machine language, forces the programmer to structure the problem solving steps in the same way the machine will execute them. When computers were first introduced, that was exactly how engineers constructed their computer programs. Since data, instructions, and memory locations are represented by 0's and 1's (See Figure 4-8), this method is very tedious and error prone and becomes nearly impossible for practical problems. In addition, every computer has its own unique machine language and because of this, *machine language programs* are not transferable between different type machines.

```
11010000 00001100 00000101 11000000
01010000 11010000 11000000 00111110
01000001 11010000 11000000 00111010
01011000 01100000 11000000 10000010
01000001 10110000 00000000 00000000
01000001 00110000 00000000 10000110
01011010 10110011 00000000 00000000
01000001 00110011 00000000 00000100
01000110 01100000 11000000 00010100
01011100 10100000 11000000 10100010
01011101 10100000 11000000 10000010
11100001 01100000 00000000 00000000
01011000 11010000 11000000 00111110
10011000 11101100 11010000 00001100
00000111 11111110

00000000 00000000 00000000 00010000
00000000 00000000 00000000 00100000
00000000 00000000 00000000 00000110
00000000 00000000 00000000 00000001
```

Fig. 4-8 Binary Object Code

4.5.2 Assembly Language

Early in computer development engineers learned to use the power of the computer to assist in the programming process. Instead of directly using binary code, the engineer developed a shorthand notational language that was easier to understand. This notational language was called assembly language. Assembly language represents each instruction with a mnemonic expression and data is represented by its equivalent decimal number. The engineer or programmer still structures the problem solving steps the same way the machine executes them; but now the computer itself is used to perform the translation from assembly language into machine language. For example, the assembly language program in Figure 4-9 will find the average of "N" number of grades.

```
STMT    SOURCE STATEMENT
 1 AVERAGE  CSECT
 2          STM      14,12,12(13)
 3 * THIS PROGRAM FINDS THE AVERAGE OF N INTEGER VALUES*
 4          BALR     12,8
 5          USING    *,12
 6          ST       13,SAVE+4
 7          LA       13,SAVE
 8 *  STANDARD LINKAGE FROM OPERATING SYSTEM *
 9          L        6,N      *REGISTER 6 USED TO INCREMENT*
10          LA       11,0     *REGISTER 11 USED FOR SUMMING*
11          LA       3,ADDR   *ADDRESS OF FIRST NUMBER
12 LOOP     A        11,0 (3) *SUM=SUM+NEXT NUMBER
13          LA       3,4 (3)  *GET ADDRESS OF NEXT NUMBER
14          BCT      6,LOOP
15          M        10,=F'1' *EXTEND SIGN BIT TO HIGH ORDER
16 *                          *REGISTER PAIR
17          D        10,N     *INTEGER PART OF AVERAGE IS
18 *                          *PLACED IN REGISTER 11 (IN HEX)
19 *                          *AND REMAINDER IS IN REGISTER 10
20          XDUMP
21          L        13,SAVE+4
22          LM       14,12,12 (13)
23          BR       14
24 SAVE     DS       18F
25 N        DC       F'6'
26 ADDR     DC       F'16,32,442,988,-26,388'
27 END
28          =F'1'
```

Fig 4-9 Assembly Language Program

This program now needs to be translated into the binary code that the machine can execute. This translation process is known as "assembly" and it is performed by another program called an "assembler". The software program written in assembly language is known as the "source" program and the binary code created by the assembler is called the "object" program or code. The assembly program (e.g., the source program in Figure 4-9) would then become the object program shown in Figure 4-8.

The introduction of assembly languages greatly simplified computer programming and resulted in an increase in productivity. Use of assembly language, however, does introduce some inefficiency in execution because the translation process introduces some overhead (additional code). The general approach to translation employed by an assembler can not always optimize the binary instructions to be as efficient as binary code written directly by a good programmer. Because the assembly language source program makes it easier to understand and to correct errors, modifying programs becomes easier and the benefits accrued from this far outweigh the inefficiency introduced by the translation process. Assembly language, however, is also computer dependent since every computer has its own assembly language and assembly language programs are not transferrable between different types of machines.

4.5.3 High Order Language

The next advance in programming occurred with the introduction of higher order languages (HOLs). HOLs use statements that are more English-like, easier to understand, more productive, easier to support, and less dependent on the computer design. Examples of higher order languages are COBOL, FORTRAN, Pascal, and Ada. Although assembly languages were an improvement over the laborious and error prone machine languages, they retained a one for one correspondence with the instruction set of a particular machine. This means that every machine instruction set has a different assembly language so when programming in that language, the programmer has to tailor the problem solving steps to the particular machine's repertoire of operations. Higher order languages broke away from this dependence. FORTRAN, which is an acronym for FORmula TRANslator, was developed for scientists and engineers and allowed the creation of mathematical algorithms and programs without the need to know the details of a particular computer ISA. The COmmon Business Oriented Language (COBOL) was created in order to help the business community manage large amounts of data. Although these languages can be used in applications other than those they were designed for, they have found their broadest application in the domain they are best suited for, namely scientific and business applications respectively. Different problem domains have created the need for new languages tailored to the peculiarities of that problem solving process. As a result of this need, hundreds of HOLs have been developed.

As with assembly language, a higher order language must be translated into a particular machine code. This process of translation is called "compiling" and the translator is a software program called a compiler. A compiler is generally a very large and complex program that translates the HOL source program to the machine executable object program. Because the object program is machine dependent, the compiler translation is also machine dependent. That means that each different computer must have its own unique compiler.

Compilers are typically designed with the flexibility for translating HOL source code to many different machines. This is done by designing the compiler with a front end and a back end. When the translation is being performed, the source program instructions are first processed by the front end to create a program in a generic intermediate assembly-like language. This front end process is the most difficult part of the translation. The second step is to process this intermediate program with the back end of the compiler. The purpose of the back end, or code generator, is to translate the intermediate code into the machine language of the "target" computer (the one that will actually execute the program). The code generation process of a compiler is similar to an assembler. To translate to different target computers, the developer has to build the unique code generator for that particular target computer and the front end, which is the toughest to develop, remains unchanged. In industry, once the first compiler is developed, subsequent compilers for different machines can be quickly constructed. An example of the previous programs (Figures 4-8 and Figures 4-9) written in the Ada programming language is shown in Figure 4-10.

4.6 Ada DESIGN

The Ada programming language was designed with three overriding concerns: program reliability and support, programming as a human activity, and efficiency [1]. The result of these concerns was the creation of a language that embodies the principles of modern software engineering practices. Applying these principles, or tools of the trade, helps one deal with two very real and difficult aspects of large-scale software developments: complexity and change.

```
with ADA_IO;
 procedure AVERAGE is
        use ADA_IO;
        -- This procedure will compute the average of
        -- N number of grades (GRADE) using integer
        -- numbers.  Declares INTEGER variables.

        N,
        GRADE,
        AVE: INTEGER;
        I: INTEGER:=0;
        SUM: INTEGER:=0;
begin
        put ("Enter the number of grades:");
        put_line ("  ");
        get (N);
        loop;
              exit when (I>=N);
              put ("Enter numerical grade:");
              get (GRADE);
              I = I + 1;
        end loop;
        AVE:=SUM/N;
        put ("AVERAGE = ");
        put (AVE);
      end AVERAGE;
```

Fig. 4-10 Ada Program

Ada was specifically designed to encourage or help the engineer or programmer develop software that is reliable and simple to maintain. In particular, Ada emphasizes program readability over ease of writing. For example, the rules of the language require that program variables be explicitly declared a particular type such as real, integer, Boolean, etc. Since the type of a variable is unalterable, compilers can ensure that the only operations performed on variables are those allowed by the rules of the Ada language syntax. This prevents the programmer from attempting to perform an illegal operation such as multiplying an integer value with a string variable. Furthermore, error-prone notations have been eliminated because the language avoids the use of cryptic encoded forms in favor of more English-like constructs. Finally, the language offers support for separate compilation of program units in a way that facilitates program development and support.

A major strength of Ada lies in its ability to perform numerous checks both during compilation and at run time. The richness of the syntax makes it more difficult for a programmer to code and compile a program, but once compiled an Ada program is more apt to run correctly than a program written in any other language. In other words, Ada enforces discipline into the programming process. This is a significant contribution to reliable software.

Concern for the human programmer was also stressed during the language design. Above all, an attempt was made to keep the language as small as possible by trying to avoid the pitfalls of excessive complexity. Ada uses simpler designs to provide language constructs that correspond intuitively to what the users would expect.

Like many other human activities, software programs are becoming larger and their development is becoming ever more decentralized and distributed. Consequently, the ability to assemble a program from independently produced software components was a central objective of the Ada design.

No high order language can avoid the problem of compiler translation efficiency. Languages that require complex and elaborate compilers; that result in the inefficient use of storage; or slow down the execution times, force these inefficiencies on all machines and on all programs. Every construct of the Ada language was examined in the light of current state-of-the-art compiler implementation techniques. Any proposed construct whose implementation was unclear or that required excessive machine resources was rejected [2].

4.7 FEATURES OF THE Ada LANGUAGE

The Ada programming language has incorporated the following features:

Abstraction means to ascribe essential properties at a given level. For example, a vehicle is an abstraction at a general level. One can provide further detail to the class of vehicles by identifying an automobile. Then one can further describe the various characteristics of an automobile such as a six cylinder engine, two door sedan and so on to describe a specific automobile. This of course can continue down to the smallest component necessary. Notice that each lower level adds detail.

Information hiding is the technique of providing only the essential information necessary for interfacing with a given unit. A specification control drawing is an example of information hiding. A drawing of this type provides information on the inputs and outputs of a particular item or device without providing any details on the internal structure of the item or device. This prevents detailed information from confusing essential information. Abstraction and information hiding assist programmers in dealing with complexity.

Modularity is the principle of logical structuring. One decomposes a design through levels of abstraction so that it has the properties of loose coupling and tight cohesion. A module can be described as an entity or unit whose internal elements are tightly bound or related (cohesion) but with light interconnections (coupling). In design, one would collect all logically related resources into one module. This is the principle of localization. Modularity and localization help the programmer to deal with complexity as well as with change. The effects of change can be better controlled or isolated through modular design and localization.

All these features contribute to reliable, maintainable, understandable, and efficient designs. They also allow the programmer to cope with the complexities of large-scale systems and the inevitability of change. Ada incorporates all of these attributes. The different concepts embraced by Ada are not new to programming languages. What is unique about Ada is that Ada is the first language to combine all these features into a single language.

There are some unique features provided by Ada that are not provided by any other production languages: the package, exception handling, tasking, and generics.

A package is an entity or collection of related objects and their operations. This collection of resources can be viewed as a wall surrounding a collection of logically related entities such as operations, data types, and related program units [2]. The package encourages and enforces the principles of modularity, localization, abstraction, and information hiding.

Exception handling provides a controlled way to exit from an abnormal event. In real-time operations, one cannot allow an abnormal event such as division by zero or a register overflow to halt the entire process. Through exception handling, abnormal events are flagged during processing and purposefully handled to prevent a catastrophic failure. This is done in one of three ways: allowing execution to follow an alternate path, restarting the operation at a controlled point, or overriding the current data with default values. Exception handling is one way of designing graceful degradation into the software.

Tasking is another principle important in real time operations. Simply stated, tasking allows concurrent or parallel processing to occur in the same or separate processors. In the real world, processes are generally concurrent. This approach to design breaks the sequential mindset and is one of the features of Ada that is most often criticized. Real-time applications require fast, reliable completion of tasks whose priorities are constantly changing. Two often heard weaknesses of Ada are mentioned here. Ada completes each task before starting another task and Ada fixes priorities at compilation time and is therefore inflexible to the changing environment or the real world [3].

Generics is a feature that reduces complexity and encourages the production of reusable code components. The concept of generics is similar to a template whereby a structure and associated operations are defined for later use in specific application. For example, a generic routine can be created for sorting. The process of sorting is the same whether one is sorting numbers, names, or objects. In other languages, one creates different sort routines for each application. With the generic capability in Ada, one routine will suffice. This reduces the need for multiple programs to perform the same task. This also aids in dealing with large complex systems by introducing common generic packages.

It is important to understand that Ada provides a means for achieving good designs because it embodies principles of good systems and software engineering. It is also important to know that the Ada programming language itself is only a small part of the process of designing sound systems. The design and development of good software requires more than just a good programming language. One must still employ sound design practices and procedures, strong configuration management, good system design tools and aids, and sufficient training on the use of these methods and tools.

4.8 TRANSITION TO Ada

It is important to understand that the Ada programming language is in a period of transition [4]. There are several factors to consider when evaluating the risk of using Ada. Foreman and Goodenough provide an Ada survival checklist in Figure 4-11. The Software Engineering Institute's publication entitled "Ada Adoption Handbook: A Program Manager's Guide" [4] provides considerable detail and suggestions for dealing with Ada in today's environment. One must understand that workarounds exist for many of the known deficiencies of Ada.

4.9 PROGRAM DESIGN LANGUAGES

One of the most important tools currently used for designing software is a program design language (PDL). The DOD's policy concerning PDLs is given in DOD Directive 3405.2, "Use of Ada in Weapon Systems" which states:

> "An Ada-based program design language shall be used during the designing of software. Use of a PDL that can be successfully compiled by a validated Ada compiler is encouraged in order to facilitate the portability of the design".

A PDL is a formal language (sometimes referred to as pseudo-code) for specifying the "blueprint" for software implementation. When Ada is used as a PDL, the properties of the language allow a PDL "blueprint" of a software design to be actually run through an Ada compiler, just like the final product, with the restriction that it doesn't execute. This allows the compiler to perform all the syntactic and semantic checks it normally performs. This is a capability not possessed by any other HOL currently in use. Several Ada-based PDLs exist. The IEEE Standard, IEEE-STD-990-1987, is one example.

* Get experience and training
* Be conscious of Ada's shortfalls
* Select a contractor with a proven track record
* Consider parallel development
* Consider development of tools and development aids
* Select a compiler based on:
  - Using more resources
  - Evaluation of performance
  - Evaluation of environment and support services
* Know your risk and manage it

Fig. 4-11 Ada Survival Checklist

A PDL is not a panacea for software development. Poor software designs can be produced using a PDL. PDLs, however, have major advantages which can make the process of software design, code, test and integration a less painful process.

4.10 REFERENCES

1. ANSI/MIL-STD-1815A, "Ada Programming Language", 22 Jan 1983.

2. Booch, Grady, "Software Engineering With Ada", Menlo Park, Ca: The Benjamin/Cummings Publishing Co.

3. Taft, Darryl K., "Revisions to Ada Standard Expected After Reviews", Government Computer News, January 22, 1988.

4. Foreman, John and John Goodenough, "Ada Adoption Handbook: A Program Manager's Guide", Software Engineering Institute Technical Report CMU/SEI-87-TR-9, May 1987.

CHAPTER 5
SOFTWARE DEVELOPMENT PROCESS

5.1 INTRODUCTION

The development of a weapon system requires integrating technical, administrative, and management disciplines into a cohesive, well-planned, and rigorously controlled process. As a critical component of a weapon system, software must be developed under a similarly disciplined engineering process. In his book "Software Engineering Concepts" [1], Richard Fairley defines software engineering as:

> "... the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates".

Barry Boehm [2] defines software engineering as a discipline that:

> "involves the practical application of scientific knowledge to the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them".

The main point is that the software development process must be scientific and disciplined. This is not different from the hardware development process. As with hardware, the goal of the software development process is to consistently produce a quality product, within schedule and cost.

With the publication of DOD-STD-2167, "Defense System Software Development", the DOD took the first step toward a standardized, systems engineering approach to software development [3]. This standard is supported by other military documents and describes a standard process and documentation for computer software development. To use this standard effectively, the program office must have a thorough understanding of the system being developed; particularly the overall system requirements and constraints. Requirements must be defined early through trade studies and prototyping. Traceability of requirements must be maintained throughout the acquisition life cycle and any requirement that cannot be traced up to a higher requirement should be modified or eliminated.

The material presented in this chapter will describe activities that occur in a "typical" program. The reader should understand that real programs seldom actually follow this "typical" profile. Phases can occur concurrently, they can be bypassed altogether, protracted, or condensed to satisfy the needs of the overall program objectives. The point to understand is that although the process is somewhat constant, its chronological occurrence is not fixed. The following sections describes the classical approach to software development.

5.2 SUMMARY OF DEVELOPMENT ACTIVITIES

Figure 5-1 presents an overview of the development activities of an integrated software and hardware system as reflected in DOD-STD-2167A.

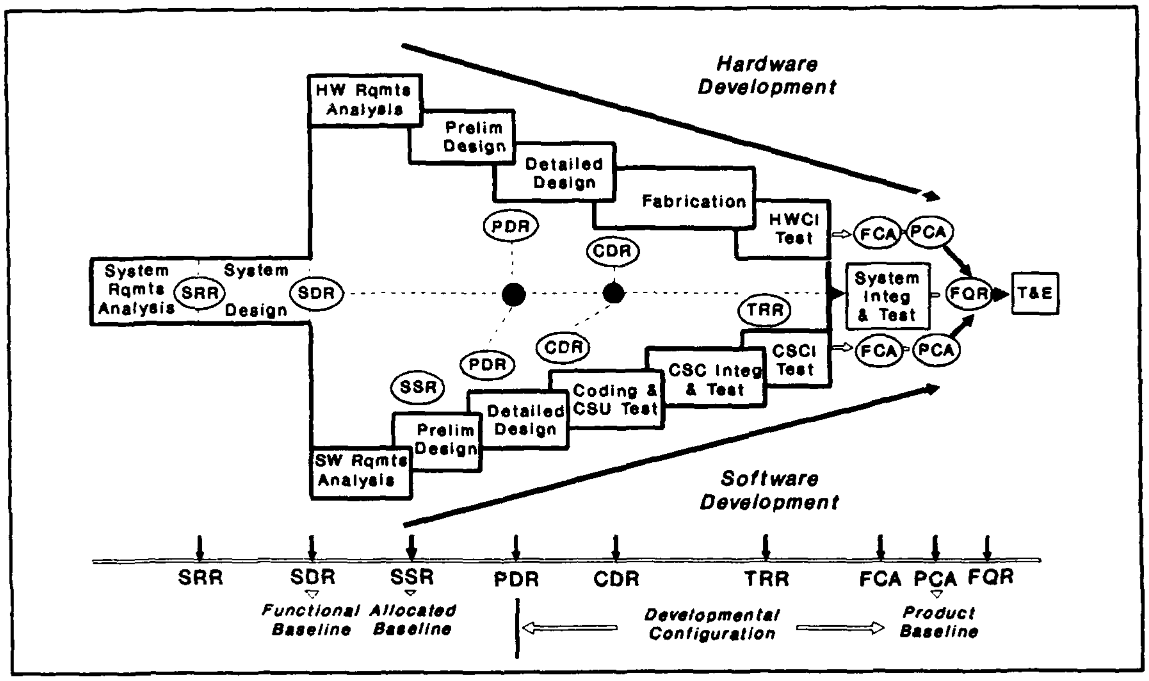


Fig. 5-1 Software/Hardware Development

All weapon system development programs begin with a determination of system level requirements. These activities occur during the Concept Exploration (CE) and the Demonstration and Validation (D/V) phases of the acquisition cycle.

The Systems Requirements Review (SRR) may be held after the initial determination of system functions (functional analysis) and the preliminary allocation of these functions to configuration items. The SRR provides an opportunity for an initial insight into the developer's direction, progress and convergence on a system configuration. The System Design Review (SDR) is a review of the overall system requirements in order to establish the functional baseline documented by the system specification. The functional baseline should allocate requirements to hardware and software configuration items.

The development of both hardware and software can begin once the Functional Baseline is established. These activities occur in the Full Scale Development (FSD) phase and are monitored through informal and formal reviews and audits as described in MIL-STD-1521B, "Technical Reviews and Audits for

Systems, Equipment, and Computer Resources". The Allocated Baseline for software should be established at the Software Specification Review (SSR). For hardware the allocated baseline is normally established at the Preliminary Design Review (PDR); but no later than the Critical Design Review (CDR).

Building of the developmental products can start once the design effort is completed. For hardware this building effort is called fabrication and for software it is called coding and testing. Testing is further subdivided into Computer Software Unit (CSU) testing and Computer Software Component (CSC) integration and testing. After the items are built, formalized testing should take place in accordance with approved test plans and procedures. A Test Readiness Review (TRR) will be conducted by the government in order to determine the developer's readiness to perform formalized acceptance testing. Completion of software testing will lead to system integration and testing. Both Functional Configuration Audits (FCA) and Physical Configuration Audits (PCA) will be conducted on both hardware and software configuration items in order to establish the respective Product Baselines. After a Formal Qualification Review (FQR) at the system level, the integrated system will be turned over to the government for operational testing as defined in the system's Test and Evaluation Master Plan (TEMP). Successful completion of this testing indicates that the product is fully defined and ready to be manufactured. For hardware, the production line would begin to assemble carbon copy items. For software, turning out copies is a trivial process since the product is complete and needs only to be duplicated on the required media for transfer to the target system computer.

5.3 SYSTEM REQUIREMENTS ANALYSIS/DESIGN

Figure 5-2 depicts the activities and products associated with the CE and D/V phases. The CE and D/V phase activities are system oriented to:

(a) Define overall project objectives;

(b) Determine project feasibility;

(c) Develop acquisition/development strategy;

(d) Establish resource cost and schedule;

(e) Define hardware/software interrelationships;

(f) Define technical and business functions and performance.

The first step in any system development is to generate the system level requirements and reflect them in a System/ Segment Specification (SSS) (Type A Specification). It doesn't

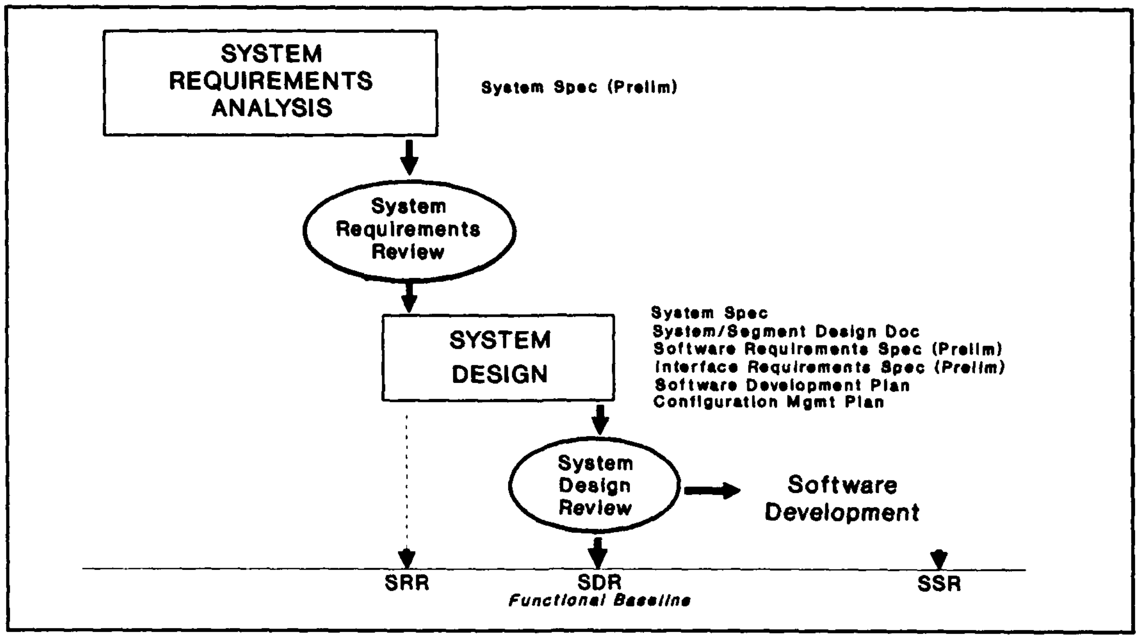

SYSTEM REQUIREMENTS ANALYSIS
System Spec (Prelim)
System Requirements Review
SYSTEM DESIGN
System Spec
System/Segment Design Doc
Software Requirements Spec (Prelim)
Interface Requirements Spec (Prelim)
Software Development Plan
Configuration Mgmt Plan
System Design Review
Software Development
SRR
SDR
Functional Baseline
SSR

Fig. 5-2 System Requirements Analysis/Design

make any difference whether it is a hardware only, a software only, or a hardware and software system; the most important and critical aspect of weapon system development is to "nail down" the system requirements. These requirements must first be finalized at the functional level, before being allocated to hardware and software.

The requirements are nailed down through a series of engineering studies and tradeoffs. These studies include:

(a) Requirements Refinement - The overall system requirements, including constraints, should be examined to identify the factors that drive requirements for computer resources. These factors may include system interfaces, interoperability, communication functions, personnel functions, the anticipated level and urgency of change, and requirements for reliability and responsive support.

(b) Operational Concept Analysis - The operational concept should be analyzed in order to determine the role of computer resources. Particular attention is paid to requirements for mission preparation, operator interface, control functions, and mission analysis.

(c) Tradeoff and Optimization - The effects of system constraints such as the operations concept, the support concept, performance requirements, logistics, availability and maturity of technology, and limitations on cost, schedule, and resources are determined. Alternative computer resources approaches are studied to:

- meet operational, interoperability, and support requirements;

- determine how the system requirements for reliability and maintainability will be satisfied;

- determine how requirements for system security will be met;

A determination will also be made regarding the suitability of standard computer languages and instruction set architectures.

(d) Risk - For each approach, the risks associated with computer resources are evaluated. Typical risk areas include compiler maturity, availability and maturity of the software support tools, loosely defined or incomplete interface definitions, and lack of adequate computer memory or throughput capability.

5.3.1 System Design

System Design begins on or about the time of the SRR. The major function of System Design is to establish the functional baseline of the system by updating and approving the system specification and the operational concept; by developing the initial subsystem/segment designs; and by further refining the systems engineering planning activities to be employed during system's development. Typical products are:

(a) System Specification;

(b) System/Segment Designs;

(c) Configuration Management Plan (CMP);

(e) Computer Resources Life Cycle Management Plan (CRLCMP);

(f) Preliminary Software Requirements Specification (SRS);

(g) Preliminary Interface Requirements Specification (IRS);

5.4 SOFTWARE DEVELOPMENT

Before undertaking a discussion of software development it will be necessary to present the following definitions:

Computer Software Configuration Item (CSCI) - A configuration item for computer software.

Computer Software Component (CSC) - A distinct part of a computer software configuration item (CSCI). CSCs may be further decomposed into other CSCs and Computer Software Units.

<u>Computer Software Unit</u> - An element specified in the design of a Computer Software Component (CSC) that is separately testable. A CSU is the lowest level of software decomposition.

Weapon system software is partitioned into CSCIs based on the program office's management strategy. Each CSCI is managed individually and follows its own development cycle. The software development cycle is defined in DOD-STD-2167A and consists of eight major activities: **Systems Requirements Analysis/Design (See Section 5.3), Software Requirements Analysis, Preliminary Design, Detailed Design, Coding and CSU Testing, CSC Integration and Testing, and CSCI Testing, Systems Integration and Testing (See Section 5.5).** These steps typically occur during FSD, although they may occur one or more times during each of the system life cycle phases [4]. This is especially true if software prototyping is performed during the Demonstration and Validation Phase. The steps are not simply linear since software development is iterative in nature and any step may be repeated many times during the course of system development. For example, problems discovered during software integration and testing may force the software designers to go back and redo the Software Requirements Analysis and all the subsequent steps.

Managing software is very similar to managing hardware; both require discipline and control in order to succeed. An important part of the control process is the formal determination of whether or not the developer is ready to proceed to the next step. This is usually determined through a series of design reviews and audits. Software reviews and audits can occur in conjunction with hardware reviews; but they do not necessarily have to. It is important that appropriate system level reviews be held at strategic intervals. This will focus everyone's (hardware and software personnel) attention on system design and leads to timely baselines for the hardware, the software, and all the interfaces. Software development has two major reviews that are separate from hardware reviews: the Software Specification Review (SSR) and the Test Readiness Review (TRR).

The SSR is a formal review of a CSCI's requirements as specified in the software specifications . A collective SSR for a group of configuration items, addressing each configuration item individually, may be held when such an approach is advantageous to the government. Its purpose is to establish the allocated baseline for preliminary CSCI design by demonstrating to the government the adequacy of the software specifications.

The TRR is a formal review of the contractor's readiness to begin formal CSCI testing. It is conducted after software test procedures are available and CSC integration testing is complete. The purpose of the TRR is to determine whether the contractor is ready to begin formal CSCI testing that can be witnessed by the government. A technical understanding must be reached on the informal test results, and on the validity and the degree of completeness of such documents as an operator's manual, a user's manual, and a computer programmer's manual.

5.4.1 Software Requirements Analysis

The first step in the software development cycle is the **Software Requirements Analysis** (Figure 5-3). The purpose of the Software Requirements Analysis is to establish detailed functional, performance, interface, and qualification requirements for each CSCI based on the System Specification. The means of testing and examining the software are also identified. During requirements analysis, prototype versions of high risk areas, user interfaces, and/or systems skeletons

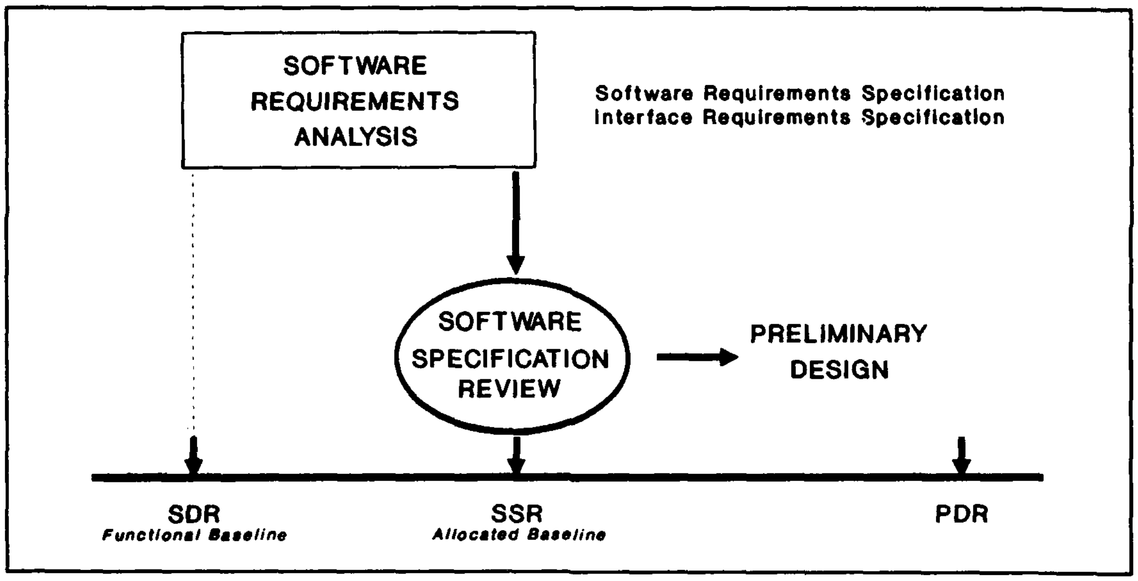

SOFTWARE REQUIREMENTS ANALYSIS
Software Requirements Specification
Interface Requirements Specification
SOFTWARE SPECIFICATION REVIEW
PRELIMINARY DESIGN
SDR
Functional Baseline
SSR
Allocated Baseline
PDR

Fig. 5-3 Software Requirement Analysis

may be partially designed and coded. Prototyping is an excellent tool for performing requirements analysis.

The developer should also identify support tools and resources, and establish timing and sizing estimates. The Program Manager must ensure that all software requirements, as reflected in the software development specifications, are traceable to the system specification and that the Software Development Plan is updated to identify the required resources, facilities, personnel, development schedule and milestones, and software tools. The developer may also customize the techniques, methodologies, standards and procedures to be used in software development.

The outputs of the Software Requirements Analysis are final versions of the software specifications, and an updated Software Development Plan. These documents will be reviewed at the SSR. The Computer Resources Life Cycle Management Plan (CRLCMP) may also be updated.

5.4.2 Preliminary Design

After the software allocated baseline is established, the developer proceeds into the **Software Preliminary Design** as depicted in Figure 5-4. Preliminary design activity determines the overall structure of the software to be built. Based on the requirements, the developer partitions the software into components and defines the function of each component and the relationships between them. Input and output relationships with external devices (such as displays and sensors) are refined according to the hardware configuration and software structure. The timing and memory budget for components are established to ensure that the software requirements can be satisfied within the hardware constraints.

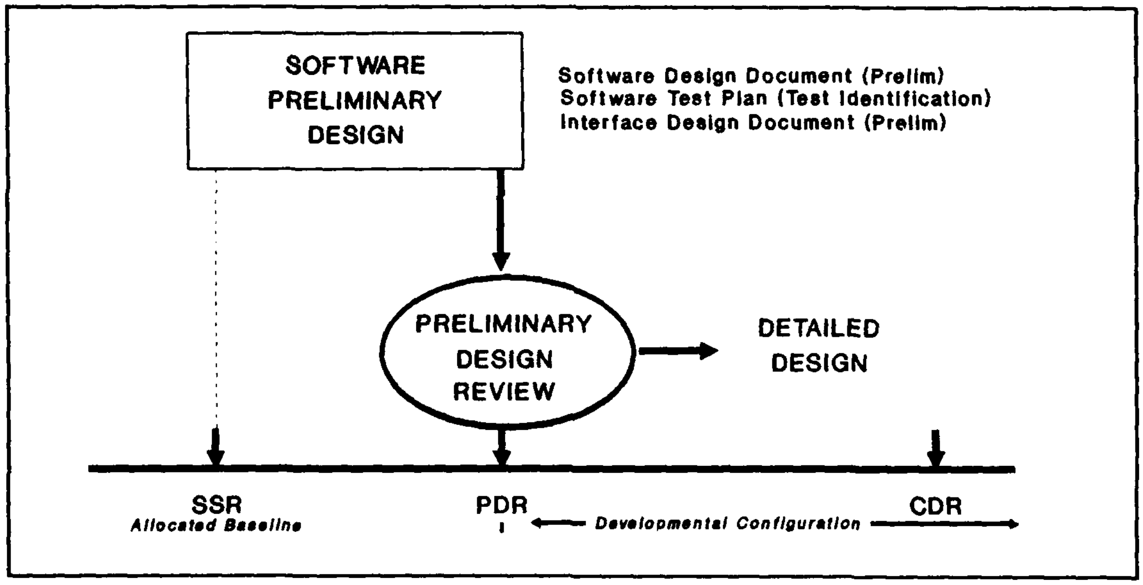

SOFTWARE PRELIMINARY DESIGN
Software Design Document (Prelim)
Software Test Plan (Test Identification)
Interface Design Document (Prelim)
PRELIMINARY DESIGN REVIEW
DETAILED DESIGN
SSR
Allocated Baseline
PDR
Developmental Configuration
CDR

Fig. 5-4 Software Preliminary Design

The developer should provide a preliminary design that insures clear traceability of requirements from the software specifications down to the software components for each CSCI. The software design is reflected in the preliminary Software Design Document (SDD) and Interface Design Document (IDD). These documents will describe the system architecture, memory and processing time allocations, interrupt requirements, timing and sequencing considerations, and input/output constraints for each software component. The developer should also generate a Software Test Plan (STP) outlining the proposed test program and establishing test requirements for software integration and testing.

The output of the contractor's efforts are preliminary versions of the software design documents and the Software Test Plan. These documents are reviewed during the PDR. Throughout the development effort, the developer will conduct informal design reviews, inspections, and walkthroughs to evaluate the progress and correctness of the design for each software component. The results of these inspections will serve as the basis for material presented at the PDR.

5.4.3 Detailed Design

The purpose of the **Detailed Design** (Figure 5-5) activity is to logically define and complete the detailed software design (not coding) that satisfies the allocated requirements. The level of detail of this design must be such that the programming of the computer program can be accomplished by someone other than the original designer. The component's function, its inputs and outputs, plus any constraints (such as memory size or response time) should be defined. Logical, static, and dynamic relationships among the components should be specified and the component and system integration test procedures generated.

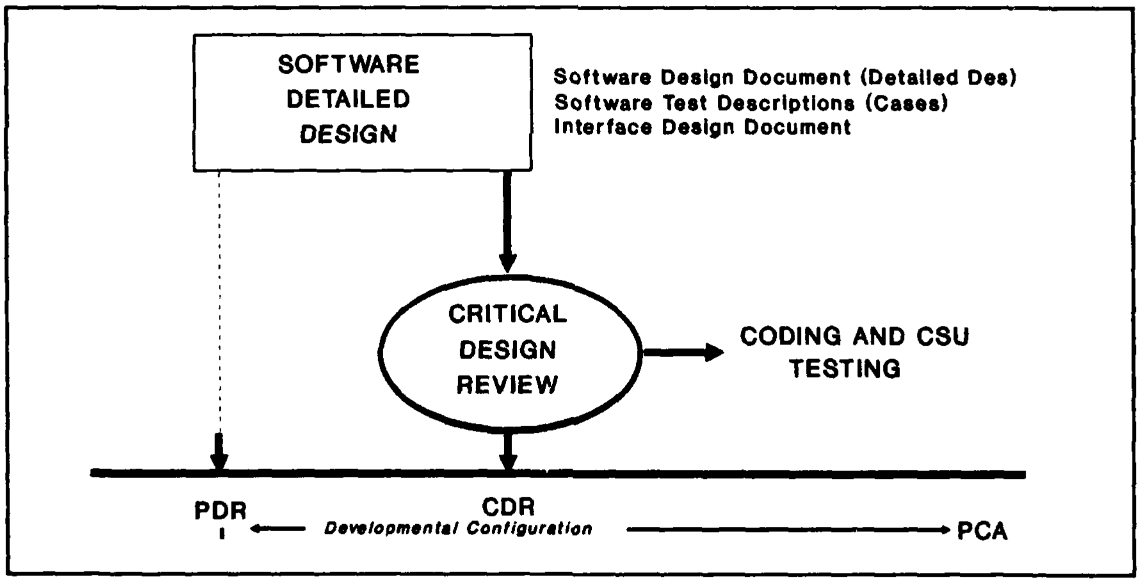

SOFTWARE DETAILED DESIGN
Software Design Document (Detailed Des)
Software Test Descriptions (Cases)
Interface Design Document
CRITICAL DESIGN REVIEW
CODING AND CSU TESTING
PDR
CDR
Developmental Configuration
PCA

Fig. 5-5 Software Detailed Design

A complete detailed design includes not only a description of the computer processes to be performed but also detailed descriptions of the data to be processed. A data dictionary is an effective way of documenting this needed design information. For software that processes or manipulates a large amount of interrelated data, the structure of the data itself should be defined.

Components which must be coded in assembly language or another "non-standard" language should be clearly defined and the reasons for the departure justified. Any special conditions that must be followed when programming the component should be similarly described and clearly documented [5]. These exceptions are normally addressed in the Software Development Plan.

During the entire design and development process the contractor should document the development of each unit, component, and CSCI in software development folders (SDFs). A separate SDF should be maintained for each unit, each component, and each CSCI. The SDFs are normally maintained for the duration of the contract and made available for government review upon request. A set of SDFs may include the following information:

(a) Design considerations and constraints;

(b) Design documentation and data;

(c) Schedule and status information;

(d) Test requirements and responsibilities;

(e) Test cases, procedures and results.

The contractor documents and implements procedures for establishing and maintaining SDFs in a Software Development Library. The library is a management tool used by the contractor to assist in developmental configuration management. It serves as a "storage house" to control access of software, documentation, and associated tools and procedures used to facilitate the orderly development and subsequent support of software [6].

A CDR is conducted at the conclusion of the detailed design. The CDR should assure that the software design satisfies the requirements of both the system level specification and the software development specifications. Following an acceptable CDR, and not before, the design should be released for coding and unit testing. This process may occur incrementally with individual releases of CSCIs.

5.4.4 Coding and CSU Testing

The purpose of programming is to translate the detailed software design into a programming language such as Ada. It is during the programming activity that listings of the source program are generated (Figure 5-6). Based on the detailed software design presented in the design specification, programming of each unit is accomplished by the assigned programmer in the specified programming language, usually Ada. As the programming of each unit is completed, the programmer examines the program for errors. Only after the programmer is satisfied that the source program correctly implements the detailed design, should the program be compiled. Compiling translates the source program to its machine executable form, the object program.

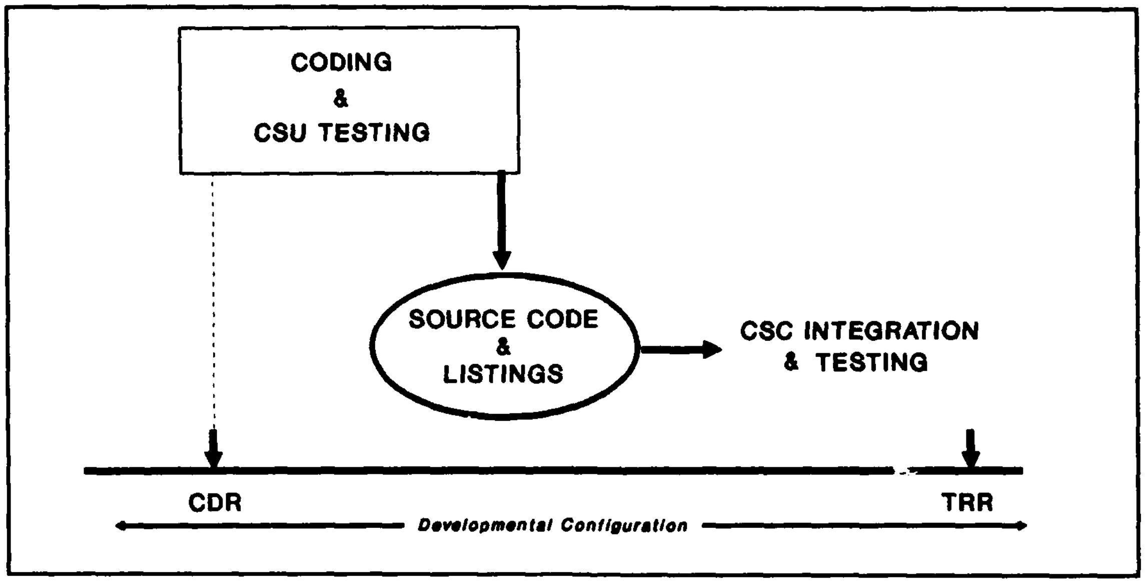


Fig. 5-6 Coding and CSU Testing

If the detailed design is in error, is ambiguous, or is not sufficiently complete to permit the programming to continue without further definition, the programmer should consult the original designer. The resolution should be documented, and all affected requirements, design, and test documentation updated accordingly.

The purpose of the unit testing activity is to eliminate any errors that may exist in the units as they are programmed. These errors may be due to mistakes by the programmer or deficiencies in the software requirements and design documentation. Usually, the test of a unit is the responsibility of the programmer who programmed the unit. Unit testing is the activity that permits the most control over test conditions and visibility into software behavior. An efficient software development effort requires rigorous unit level test so that most errors are detected before CSC Integration and Test.

Besides producing the source and object code and their respective listings, the contractor develops and records in software development folders the informal test procedures for each unit test as well as the test results. The contractor will usually conduct informal code inspections or walkthroughs on each coded unit and component during several stages of its development. There are no formal reviews scheduled during this step of the development cycle.

5.4.5 CSC Integration and Testing

Once the software is programmed and each unit and component is tested for compliance with its design requirements, the contractor should begin CSC Integration and Testing as illustrated in Figure 5-7. The purpose of CSC Integration and Testing is to combine the software units and components that have been independently tested into the total software product and to demonstrate that this combination fulfills the system design. The integration is done in a phased manner with only a few components being combined at first, additional ones added after the initial combination has been tested, and the process repeated until all components have been integrated. The phasing of this integration should be based on the functional capabilities that can be demonstrated by specific groups. There may be some overlap with the previous step in that some software components may be ready for integration while others are still being programmed.

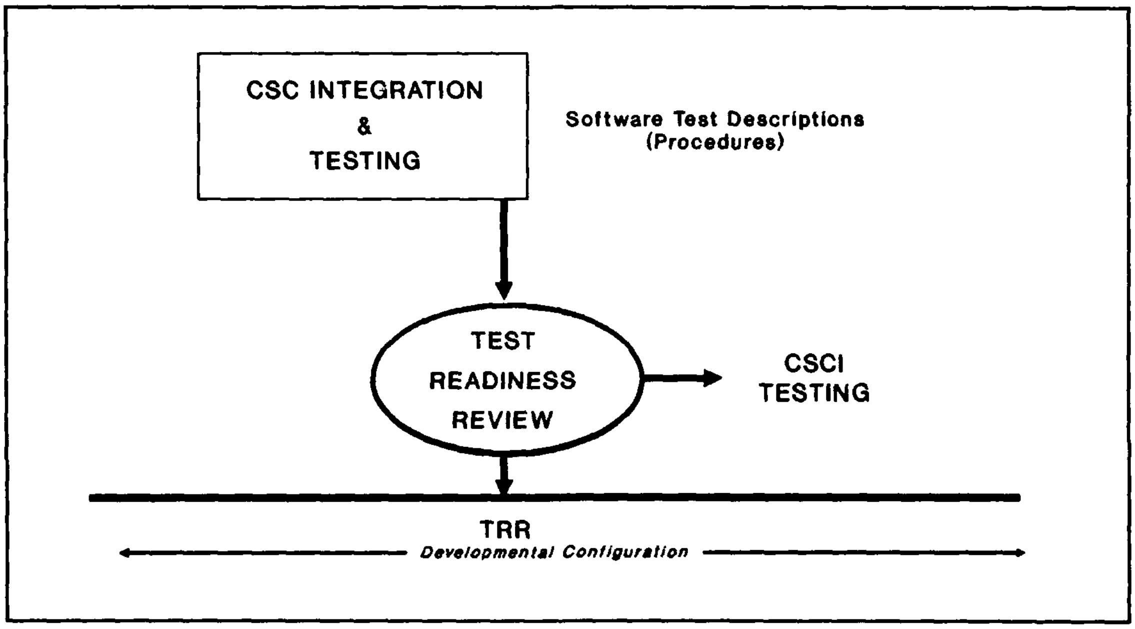

CSC INTEGRATION & TESTING
Software Test Descriptions (Procedures)
TEST READINESS REVIEW
CSCI TESTING
TRR
Developmental Configuration

Fig. 5-7 CSC Integration and Testing

Most testing performed during Coding and CSU Testing, and CSC Integration and Testing is called "informal testing". This term doesn't imply that the testing is "casual" or "haphazard", but instead implies that the testing doesn't require government approval. Some formal testing may be accomplished during these steps, but most formal testing is usually accomplished during the next step.

5.4.6 CSCI Testing

After completion of a successful TRR, the contractor will proceed with CSCI Testing (Figure 5-8), the last step of the software development cycle. The purpose is to perform formal tests, in accordance with the software test plans and procedures, on each CSCI and to establish the software Product Baseline. Testing during this step is intended to show that the software satisfies the Software Requirements Specification and the Interface Requirements Specification.

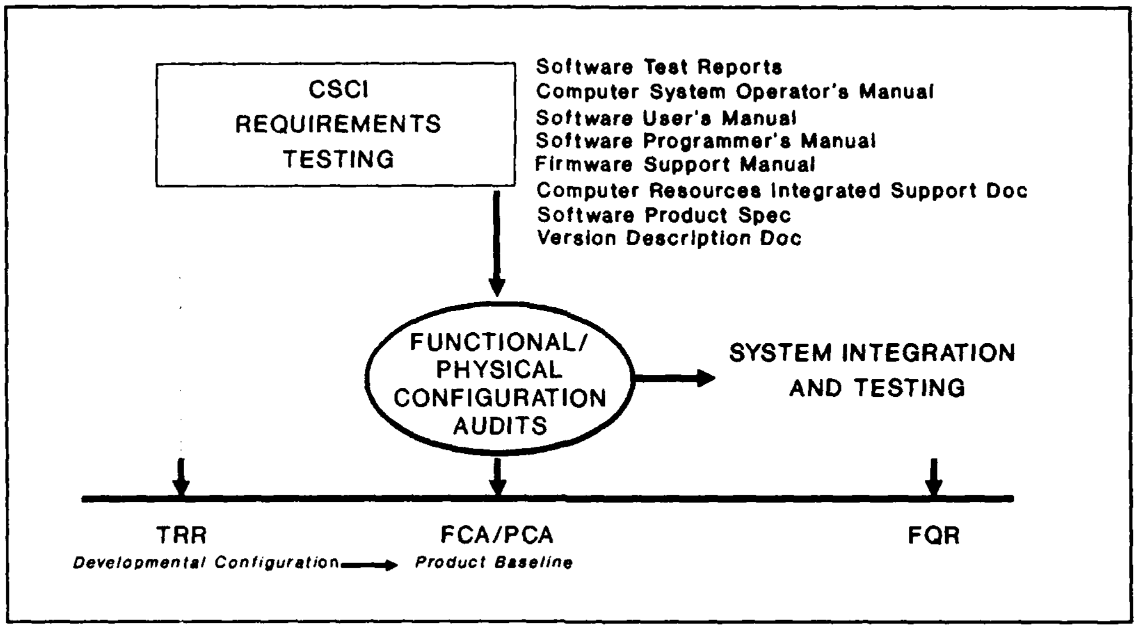

CSCI REQUIREMENTS TESTING
Software Test Reports
Computer System Operator's Manual
Software User's Manual
Software Programmer's Manual
Firmware Support Manual
Computer Resources Integrated Support Doc
Software Product Spec
Version Description Doc
FUNCTIONAL/ PHYSICAL CONFIGURATION AUDITS
SYSTEM INTEGRATION AND TESTING
TRR
FCA/PCA
FQR
Developmental Configuration
Product Baseline

Fig. 5-8 CSCI Testing

Throughout CSCI testing, the contractor should be updating all previous software documentation, analyzing test data, generating the Software Test Reports (STR), and finalizing the Software Product Specification (SPS) (C-5 *Specification*). This will be the basis for the software Product Baseline normally established at the PCA, which may immediately follow, or be conducted concurrently with, the FCA for a software only development. Normally, the PCA occurs after the software is released for integration and testing with the system following the software FCA as illustrated in Figure 5-8. During the software FCA the government verifies that the CSCIs perform in accordance with their respective requirements and interface specifications by examining the test results and reviewing the operational and support documentation. The PCA is the formal technical examination of the as-built software product against its design. This includes the product specification and the as-coded documentation.

The typical outputs of the contractor's efforts in CSCI Testing are the Software Test Report (STR), operational and support documentation such as the Computer System Operator's Manual (CSOM), the Software Users Manual (SUM), the Software Programmer's Manual (SPM), the Firmware Support Manual (FSM), the Computer Resources Integrated Support Document (CRISD), the Version Description Document (VDD), and the Software Product Specification (SPS). Except for updates and/or revisions, all deliverable documentation should be completed at this time. Figure 5-9 contains a listing of the standardized software documentation, as defined in DOD-STD-2167A, that may be required for software development programs.

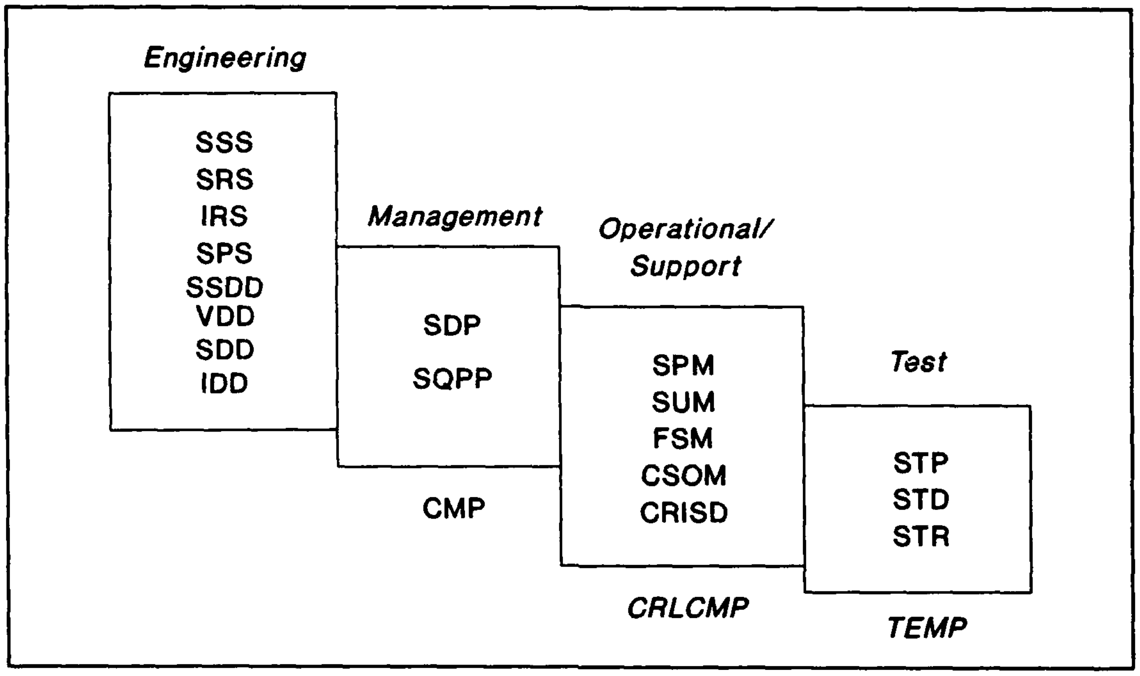


Fig. 5-9 Software Products

5.5 SYSTEM INTEGRATION AND TESTING

The purpose of System Integration and Testing is to ensure that the developed software works with the system in the environment that it was designed for (Figure 5-10). The system is turned over to the government after an acceptable Formal Qualification Review (FQR). The FQR is a system-level review that verifies that the actual system performance complies with the system requirements. For computer resources, it addresses the aspects of the software and hardware performance that have been tested after the FCA and PCA. A successful FQR is predicated on a determination that the system meets the specified requirements in the hardware, software and interface specifications.

The contractor's role will diminish significantly subsequent to the FQR. Contractor configuration control of the software should terminate once the product baseline is approved and the government assumes responsibility. All updated documentation, source and object code listings, and all other items stipulated in the contract will be delivered to the government. The government will then assume configuration control responsibility. The contractor, however, will be available to support the government's test and evaluation efforts and to conduct any required acceptance tests..

5.6 TAILORING

The purpose of tailoring is to reduce the overall costs of an acquisition, primarily by reducing the amount and type of documentation being delivered by the contractor and by

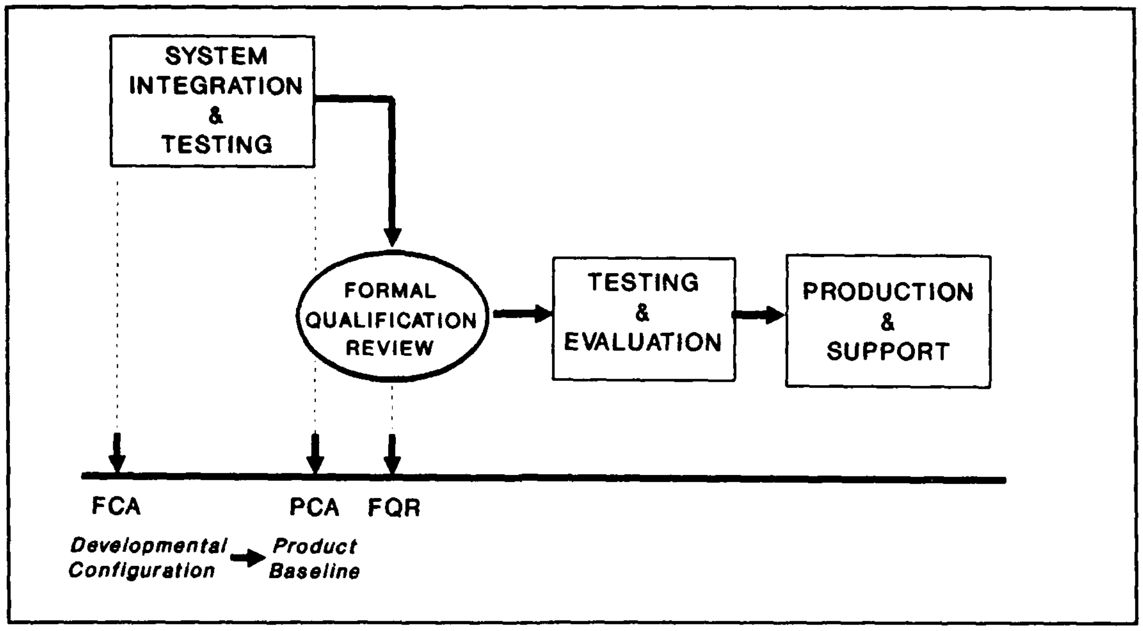

SYSTEM INTEGRATION & TESTING
FORMAL QUALIFICATION REVIEW
TESTING & EVALUATION
PRODUCTION & SUPPORT
FCA
PCA
FQR
Developmental Configuration
Product Baseline

Fig. 5-10 System Integration and Test

eliminating redundant or unnecessary testing or procedures. Some questions whose answers will provide tailoring guidance are:

(a) Is all of the documentation described in DOD-STD-2167A necessary?

(b) What documentation is already available?

(c) Is it cost-effective to modify it?

(d) Is the contractor's format acceptable?

(e) How many copies are actually needed?

(f) How can DOD-STD-2167A be tailored?

(g) Is a formal design review necessary for each CSCI?

(h) How should they be scheduled?

DOD-STD-2167A states that the tailoring process for this standard entails the deletion of non-applicable requirements. But how does a program manager determine which requirements are not applicable? Figure 5-11 illustrates the tailoring process.

Most tailoring is implemented through the statement of work (SOW). A thorough understanding of requirements (functional, performance, test, documentation) is required in order to properly tailor the standards and specifications.

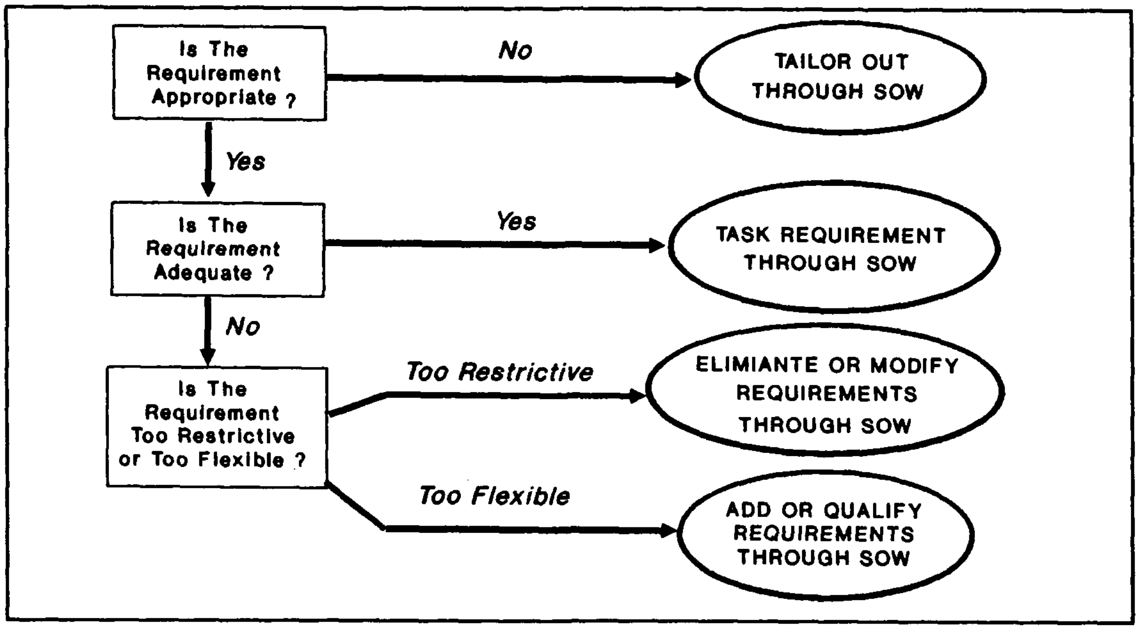

Is The Requirement Appropriate ?
No
TAILOR OUT THROUGH SOW
Yes
Is The Requirement Adequate ?
Yes
TASK REQUIREMENT THROUGH SOW
No
Is The Requirement Too Restrictive or Too Flexible ?
Too Restrictive
ELIMIANTE OR MODIFY REQUIREMENTS THROUGH SOW
Too Flexible
ADD OR QUALIFY REQUIREMENTS THROUGH SOW

Fig. 5-11 Tailoring Process

The first step is to ask if the requirement is appropriate? If not, then tailor it out through the SOW. If the requirement is appropriate, then ask if the requirement is adequate? If it is, then impose the requirement through the SOW. If the requirement is not adequate, ask if the requirement is too restrictive or too flexible? If it's too restrictive, delete it or modify it in the SOW. If it's too flexible, add to or modify the requirement in the SOW. Use careful judgement when tailoring a program. Don't tailor areas arbitrarily simply to reduce program costs because in the long run, this may increase life cycle costs.

5.7 SUMMARY

Software that is part of a weapon system is managed by partitioning into CSCIs. Each CSCI is managed individually and follows its own software development cycle. Software development activities can be broken down into six steps; any of which can be repeated as many times as necessary during the development cycle. These six steps are Software Requirements Analysis, Preliminary Design, Detailed Design, Coding and CSU Testing, CSC Integration and Testing, and CSCI Testing. These steps typically occur during the Full Scale Development Phase.

DOD-STD-2167A is the approved standard to be used by DOD agencies for software development. It is to be used in conjunction with DOD-STD-2168, "Software Quality Program Plan". These two standards are not intended to discourage the use of any particular software development method, but instead, aid the Program Manager in developing and maintaining quality software. They should be used throughout the acquisition life cycle and tailored according to system needs.

It is especially important to develop the product as a system. Never lose sight of the fact that hardware and software development are intimately related. Although they are developed in parallel, software is almost always in the critical path and it is up to the Program Manager to insure proper integration of the two through carefully planned reviews and audits. The talents of an independent verification and validation (IV&V) activity may be used to aid in this process.

5.8 REFERENCES

1. Fairley, Richard E., "Software Engineering Concepts", Tyngsboro, Mass: McGraw Hill Book Co., 1985.

2. Boehm, Barry, "Software Engineering Education: Some Industry Needs", in Software Engineering Education: Needs and Objectives, Edited by P. Freeman and A. Wasserman, Springer-Verlag, Berlin 1976.

3. DOD-STD-2167A, "Defense System Software Development", 29 February 1988.

4. DOD Directive 5000.29, "Management of Computer Resources in major defense Systems", 26 April 1976.

5. Rubey, Raymond J., "A Guide to the Management of Software in Weapon Systems", 2nd Edition, March 1985.

6. Ferens, Daniel V., "Mission Critical Computer Resources Software Support Management", Air Force Institute of Technology, Wright-Patterson AFB, Ohio, First Edition, May 1987.

CHAPTER 6
SOFTWARE TEST AND EVALUATION

Software test and evaluation is one of the most difficult, frustrating, and expensive activities that is performed during system development. Unfortunately, it is also one of the most misunderstood functions of the entire system acquisition cycle. Before discussing the details of software testing, let us review the software development life cycle and see how the software test process fits in.

6.1 TEST PLANNING

Test and Evaluation (T&E) planning is initiated at the inception of the development process. During the Concept Exploration (CE) Phase the initial draft of the Test and Evaluation Master Plan (TEMP) is developed. The TEMP is the basic planning document for all T&E related to a particular system acquisition and is used by the Office of the Secretary of Defense (OSD) and all DOD components in planning, reviewing, and approving all T&E activities. The TEMP provides the basis and authority for all other detailed T&E planning documents.

The TEMP addresses two types of computer resources: system support and embedded computer resources.

6.1.1 System Support Computer Resources

System support computer resources include all the government and contractor planned software and computer resources, required to fully test the overall system. These test resources include [1]:

<u>Test Support Equipment</u> - All unique or modified test support equipment required to conduct the planned test program including any special calibration and software requirements.

<u>Threat Systems</u> - All threat simulators against which the system will be tested including the number and timing requirements.

<u>Simulators, Models, and Testbeds</u> - All system simulations required including computer-driven simulation models and hardware-in-the-loop testbeds identified by specific test phase.

<u>Special Requirements</u> - All non-instrumentation capabilities and resources such as special data processing or databases.

For all of these test resources the system requirements are compared with existing and programmed capabilities in order to identify any major shortfalls.

6.1.2 Mission Critical Computer Resources

The initial draft of the TEMP will include a preliminary Software Test and Evaluation Plan as an appendix. This plan describes the anticipated software testing necessary to demonstrate the ability of the mission critical computer resources to achieve the system objectives. The types and extent of testing will be discussed in greater detail later in this chapter. Detailed information on the TEMP may be found in DOD Directive 5000.3-M-3, "Test and Evaluation Master Plan (TEMP) Guidelines".

During the Demonstration and Validation (D/V) Phase, the TEMP is updated to reflect further refinements in the objectives and evaluation criteria of the weapon system computer resources and to include plans for Developmental Test and Evaluation (DT&E) and Operational Test and Evaluation (OT&E). As part of contractor involvement, a Software Development Plan (SDP) is generated along with a Software Test Plan (STP). The contractor developed STP must reflect the overall Software Test and Evaluation Plan as stated in the TEMP. Along with the SDP, they become the basic documents governing the conduct of the mission critical computer resources development and test activities.

Once the Software Requirements Specification (SRS) and the Interface Requirements Specification (IRS) have been generated and approved, the preliminary software design is initiated (Figure 6-1). Preliminary design is the development of an overall skeletal structure or architecture for the software. The overall structure is defined to include such things as the types, names and number of software modules; their calling sequence and their input and output parameters; their approximate execution times; and other pertinent relationships that should exist between the various modules. Since Ada modules and packages can be compiled without the requirement that lower level code be available, preliminary design using an Ada-based Program Design Language (PDL) is very valuable.

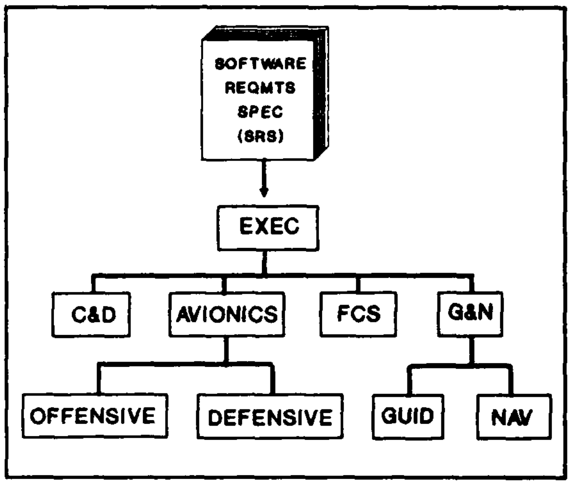


Fig. 6-1 Preliminary Design

Official government approval will be provided once a preliminary design has been completed and a Preliminary Design Review (PDR) has been held. The PDR should be approached with care and preparation since its completion is a signal to the contractor to proceed with the detailed design.

During the detailed design (Figure 6-2), the overall architecture developed during the preliminary design is fleshed out with detailed algorithms and logic implementation details. In the past, pseudo-code (English-like programming statements) was often used in this phase. DOD policy, however, now requires that the detailed design also be developed using Ada or an Ada based PDL. Once again, use of Ada as a design language will be immensely useful because the actual coding of the detailed design will be a natural follow-on to the detailed design. During both the preliminary and detailed design phases, software "test" practices such as inspections and walk-throughs (which will be discussed in detail later in this chapter) will be immensely useful in finding errors or inconsistencies in the overall system design. The Critical Design Review (CDR) officially terminates the detailed design phase.

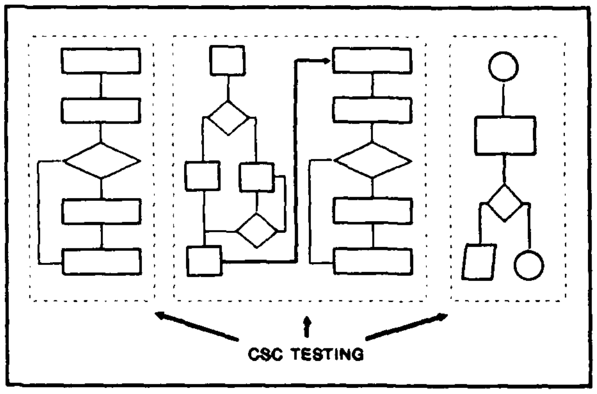


Fig. 6-2 Detailed Design

After successful completion of the CDR, coding initiates the process of building sub-modules. These submodules are progressively combined with other submodules to form larger and more complex modules and blocks of software (Figure 6-3). With each submodule or unit coded, individual and group computer software component (CSC) testing can start. This testing is usually informal in nature and is performed by the programmer. Formalized testing, as described in the Software Development Plan and the Software Test Plan, can begin after the completion of coding and informal programmer testing. The contractor can conduct his own formalized testing but any formal testing which is to be witnessed by the government, doesn't begin until a formal Test Readiness Review (TRR) has been conducted. The purpose of the TRR is to determine whether the contractor has completed his own testing and has the resources and the plans and procedures to formally demonstrate to the government that the software works as an entity.

```
task WARNING is
   entry FAULT_IN_SENSOR;
   entry OUT_OF_LIMITS (ON_SENSOR: in SENSOR_NAME;
end WARNING;

- -

task RECORDING is
   entry LOG_STATUS (OF_SENSOR: in SENSOR_VALUE;
                     WITH_VALUE: in SENSOR_VALUE;
                     WITH_STATE: in SENSOR_STATE);
end RECORDING;
```

Fig. 6-3 Coding

Software integration testing exercises a single Computer Software Configuration Item (CSCI). This type of testing usually requires a dedicated mainframe computer since other software or subsystem simulations will have to be used (Figure 6-4). Sometimes the actual target computer (or an engineering model) may be used to test the software. The actual hardware, however, typically is not available at this time and a simulation or emulation is substituted. One of the primary purposes of this detailed software system testing is to ensure that the software is inherently sound and that it demonstrates the potential for performing its function once it is married to the system hardware in a test setup called a hot bench.

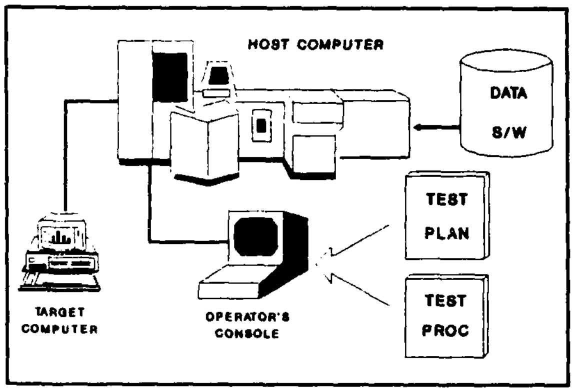


Fig. 6-4 Software Integration Testing

Hot bench integration or testing is perhaps one of the most frustrating parts of software T&E (Figure 6-5). Once the software is married to the actual hardware, the difficult part begins. The first step is to establish a hot bench or Systems Integration Lab (SIL), as it is sometimes called. The next step is to populate the SIL with actual black boxes, cable runs, power supplies, displays and system computers all configured as closely as possible to the final article. Unfortunately, a laboratory environment can only approximate the real world since a laboratory is a relatively benign environment. In spite of this, intermittent, spurious, and often mysterious problems seem to arise. These and other problems will be discussed later in the chapter.

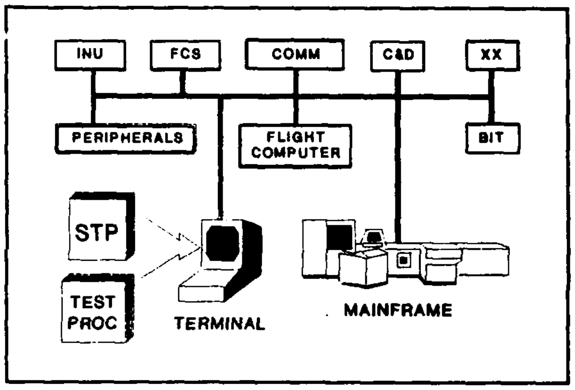


Fig. 6-5 Hot Bench Integration

DT&E and OT&E can begin once hot bench testing is successfully completed (Figure 6-6). Although the two tests are sometimes combined because of schedule constraints, the objectives of the two test are different and the kinds and amount of data required by each are different. Usually the level of detail required for DT&E is much more than that required for OT&E. This means that the hardware and software testing required for DT&E is usually more stringent than that required for OT&E.

6.2 COST OF SOFTWARE FIXES

Before discussing the details of software testing, it would be instructive to examine the cost of removing errors from software. Figure 6-7 shows the cost of a typical software development project broken down by the various phases of development [2]. As one can see, the cost of finding and correcting a software problem in the early phases is insignificant when compared to the cost of finding and correcting that same problem once the software has been delivered. Although these are not absolute numbers that apply all development projects, the message is clear; "Spend more time up front finding errors and you will reduce your overall cost".

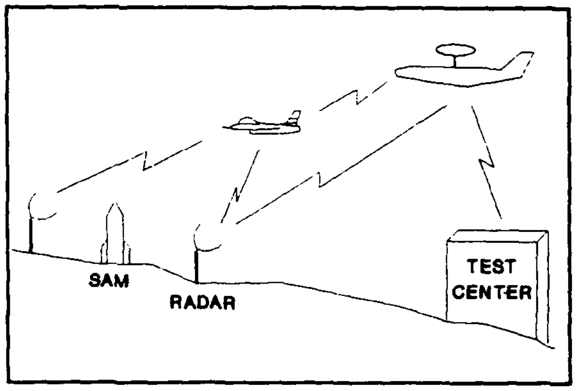


Fig. 6-6 DT&E/OT&E Testing

The question that needs to be asked is "Why does it cost so much less to find and correct errors in the earlier stages of software development?" There are several reasons that have been put forth:

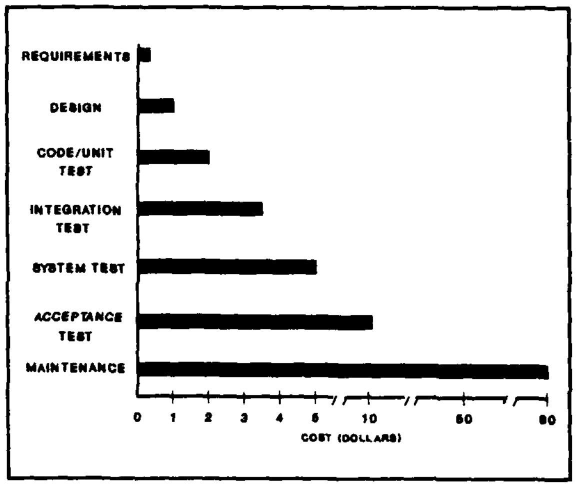


Fig. 6-7 Cost of Software Fixes

(a) At the beginning the pressure is less intense and, since there is little or no code to examine, there is more time to look for the problem.

(b) The amount of paperwork and the amount of detail to be examined is much less. Top-level design requirements, interfaces, and test requirements are being examined without regard to implementation details.

(c) Errors introduced at the top level, such as in requirements or overall system design, will be propagated manyfold into the detailed design and coding of the discrete software components.

(d) Software programmers seem to undergo a psychological change once code has been generated and computer based testing has begun [3]. They seem to be less sensitive about their mistakes when dealing with requirements and design considerations than they are about errors in coding.

(e) Once computer based testing begins, major resources are tied up in testing. Mainframe computers, weapon system hardware, technicians, and system analysts can escalate cost very quickly; hence, once a problem is found there is a tremendous psychological pressure to correct it as soon as possible. Unfortunately, the amount of paperwork involved in the correction procedure is immense and the impact of one software error usually has propagated throughout other modules. Corrections are not always complete or totally accurate [3].

In summary, it is very clear that the greatest return on the dollars invested in finding and correcting software problems occur during the early stages of software development: the requirements and design phases.

6.3 SOURCES OF SOFTWARE ERRORS

If one examines where the typical software errors occur, it is somewhat surprising to learn that they also occur mostly in the early stages of software development. As can be seen from Figure 6-8, about 40% of all software errors are attributed to problems in specification. Twenty-eight percent are due to incomplete or erroneous specifications and 12% are due to intentional deviations from the specification. Violations of programming standards contribute another 10%. Errors due to coding and programming mistakes (i.e., erroneous data accessing, erroneous logic, erroneous computations, improper interrupts, wrong constants and data values) comprise only about 38% of all errors [2].

CATEGORY	PERCENTAGE
INCOMPLETE/ERRONEOUS SPEC	28
INTENTIONAL DEVIATION FROM SPEC	12
VIOLATION OF PROGRAMMING STDS	10
ERRONEOUS DATA ACCESSING	10
ERRONEOUS LOGIC	12
ERRONEOUS COMPUTATIONS	9
INVALID TESTING	4
IMPROPER INTERRUPTS	4
WRONG CONSTANTS/DATA VALUES	3
DOCUMENTATION	8
TOTAL	100

Fig. 6-8 Sources of Software Errors

Comparison of this figure with the previous figure is very revealing. It tells us that the bulk of the errors occur during those phases of software development when errors are the least expensive to fix. The conclusion is very obvious: the more time and effort that is put into requirements definition and design, the fewer the mistakes and the cheaper it will be to correct them. The more you pay now, the less that you will pay later.

6.4 TYPES OF TESTING

Software testing can be broken up into three general categories: human testing, software only testing, and integration testing.

6.4.1 Human Testing

Human testing is defined as an informal, non-computer-based method of examining computer program architectures, designs and internal and external interfaces for the express purpose of determining how well they reflect overall system requirements [3]. Human testing is comprised of inspections, walk-throughs, desk checking, peer ratings, and design reviews.

6.4.1.1 Inspections

During an inspection, the programmer explains to a group of three or four peers the overall approach, rationale, choice of algorithms, logic, and overall module structure of the program. The purpose is to ensure correctness and consistency in structure, coding conventions such as variable definitions and use, programming standards and procedures, and overall unity of design. At the end of the inspections, which usually last about two hours, all errors, inconsistencies, and omissions are listed and given to the programmer for correction. Under no circumstances is the list of errors ever allowed to be reviewed by the programmer's supervisor or anyone else. The purpose of the inspection is to improve the final product in a non-threatening environment and not to appraise the programmer's performance. Statistics are also collected in order determine the quality of the product and the progress being made in the development process.

6.4.1.2 Walk-throughs

During a walk-through, the group of three or four peers come prepared with test cases so that they can "play computer" and mentally step through the design and the logic flow. In a similar fashion to inspections, all errors, inconsistencies, and omissions are summarized and given to the programmer for correction. The results are also confidential and are never seen by anyone outside the group, including the programmer's supervisor. The climax of both inspections and walk-throughs is a meeting of the minds between all the participants [3]. Statistics may once again be gathered.

6.4.1.3 Desk Checking

Desk checking is the least productive of all human testing since it involves the programmer sitting at his desk and reviewing his work. It is human nature to miss errors that one has committed. Desk checking, however, can be performed individually at any time and without the need for convening a meeting with other individuals. It is better than doing nothing at all.

6.4.1.4 Peer Ratings

Peer ratings involve a group of programmers reviewing each other's work. Each programer submits 1 or 2 modules to his peers and everyone anonymously reviews each others work. Results are tabulated and passed around. Like inspections and walk-throughs, peer reviews are very beneficial in improving commonality, overall consistency, and program integrity.

There are other benefits to inspections, walk-throughs, and peer ratings beside the obvious one of finding errors and inconsistencies. The participants themselves benefit because they are exposed to other programming styles and new techniques which they may want to adapt for their own use.

6.4.1.5 Design Reviews

PDRs and CDRs are not normally considered by the government as part of the test process. This is unfortunate because the contractor considers the completion of the PDR as a signal that the preliminary design is acceptable and that he can proceed with the detailed design. Likewise, completion of the CDR is the contractor's signal to begin coding and checkout. Very often the government fails to really do a thorough review of all the documentation submitted prior to either the PDR or the CDR. Because of this, the actual PDRs and CDRs may be superficially conducted and the contractors may proceed into detailed design without a thorough review of the overall design and without assurances that the overall system requirements are actually reflected in the design.

There is no immediate solution to this problem since it is often due to the lack of adequate manpower. Documentation review prior to design reviews may always be inadequate. Alternatives are to thoroughly review only critical portions of the design, thoroughly review only the module interfaces, or augment the engineering staff with outside help. If the funding is available, an Independent Verification and Validation (IV&V) contractor can be used. Outside consultants can be brought in or help can be sought from the various military a labs or software support agencies throughout each service.

Another solution is to hold incremental PDRs and CDRs. By spreading out the reviews the available staff can pay more attention to details. This of course has to be weighed against overall schedule slippage. One must remember, however, that an incompletely reviewed software design is guaranteed to introduce schedule slippage anyway. As the commercial states, "You can pay me now, or you can pay me later".

6.4.1.6 Benefits of Human Testing

Human testing is a very productive undertaking. In his book, The Art of Software Testing, Myers [3] discusses the following positive qualities of human testing:

(a) Experience has shown that it is quite effective in finding errors and it should be used on every programming effort;

(b) Since it is usually applied between the start of design and the beginning of computer based testing, it substantially contributes to productivity and reliability;

(c) It leads to finding 30 to 70% of the design and coding errors;

(d) There is a higher probability of proper error correction since they are found early in the development phase. Programmers tend to make more errors correcting errors found during computer-based software testing;

(e) It results in errors being found in clusters or batches as opposed to computer-based testing which finds errors one at a time;

(f) It lowers the cost of software testing since the costs involved are a few programmer's man-hours as opposed to the computer resources and the large number of personnel involved in computer-based testing.

In summary, human testing provides the highest return on your investment of valuable software test and evaluation dollars.

6.4.2 Software Only Testing

Software only testing is defined as that testing that is performed solely for the purpose of determining the integrity of the software when it is tested as an entity. In other words, one wants the assurance that the software works before it is married to the hardware. The reason for this is that software is usually integrated with newly developed hardware which has its own maturation problems. Integrating untested software with unproven hardware makes it very difficult to determine where the problems lie: hardware or software.

6.4.2.1 Black Box Testing

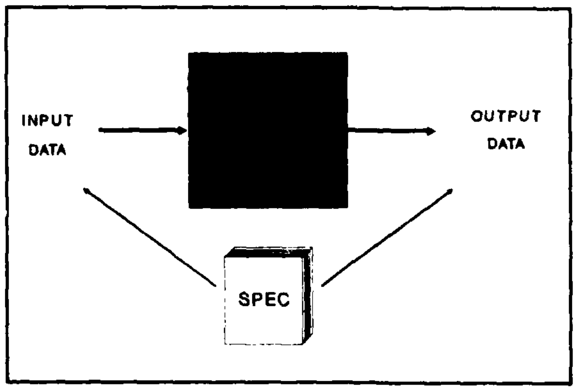


Fig. 6-9 Black Box Testing

Black box testing entails testing a particular software unit without any knowledge about the internal structure or logic of that unit (Figure 6-9). Various test cases are generated based on the specification and the requirements of the unit. The correctness of the internal structure of that software unit is then determined by the output data generated in response to the input data. This is done by designing the test cases with two types of input data. The first type of input data falls within the boundaries expected by the software unit and the second type of data falls outside these boundaries. The reason for the latter is that oftentimes it is those parameters (i.e. invalid) that do not fall within the expected limits that cause the biggest problems. It is an old dictum in software development that ensuring that the software doesn't do the unexpected is as important as ensuring that it does what it is supposed to do.

The primary objective of software testing is to demonstrate performance and reliability. Often it is useful, if not essential, to approach software testing (particularly at the lower levels) with the attitude of "Let's try to break it". This is to demonstrate the ability of the software to recover from abnormal events or to degrade in a graceful or controlled manner. Since complete and exhaustive testing of any practical sized software is impossible, one must develop an acceptable level of maturity in the software. This is often measured by the number of faults that are occurring during test. As software matures, the faults discovered decrease and, as a minimum, become controllable (i.e., the configuration is stable).

The reason that exhaustive testing of software is a practical impossibility is that the number of valid and invalid test cases are infinite. For example if a certain input parameter should fall within the values of 1 and 10, truly exhaustive testing would test all the values between 1 and 10. This is clearly an impossibility since there are an infinite number of valid values in between these two limits. Likewise, exhaustive testing of all invalid parameters is also impossible because these are also infinite.

To summarize, software of any practical size can never be exhaustively tested and can never be guaranteed to be totally error free. Because of this, the goal of software testing is to demonstrate a certain level of performance so that there are reasonable assurances that the software performs according to the specification. How reasonable the assurances are is a simple matter of economics, how much can one afford to test?

6.4.2.2 White Box Testing

Unlike black box testing, white box testing is concerned with the internal structure and logic of the software unit. The test cases are generated using a listing of the code as opposed to the requirements specification (Figure 6-10). Generation of the input data, however, is very similar to the generation of input data for black box testing. Both valid and invalid input data are generated for the test cases and the output data is analyzed to determine the correctness of the computer code. Exhaustive white box testing is also prohibitive for the same reasons that exhaustive black box testing is impossible.

Exhaustive white box testing, however, does have another dimension other than all the possible valid and invalid input parameters. Since the tester has knowledge of the internal logic and program structure, a goal would be to test all possible logic paths that exist within the code. Although all paths can be executed, it is impossible to test all combinations. As an example, consider the logic flow of Figure 6-11 which shows a simple logic sequence that contains two loops that execute up to 12 times. This sequence contains 10 raised to the 20th power different finite paths. If one could test one logical path every nanosecond (trillionth of a second), it would take 4000 years to test all the different paths. Using a real example, it would take over 60,000 years to perform a similar type of exhaustive test on the Titan III missile guidance software.

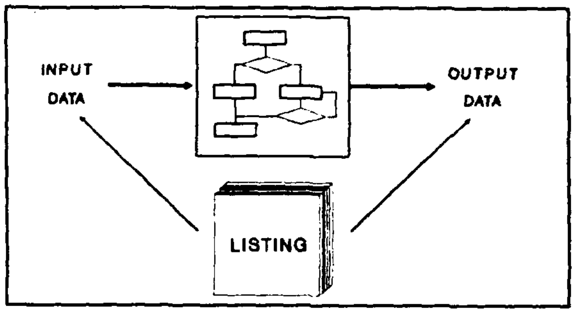
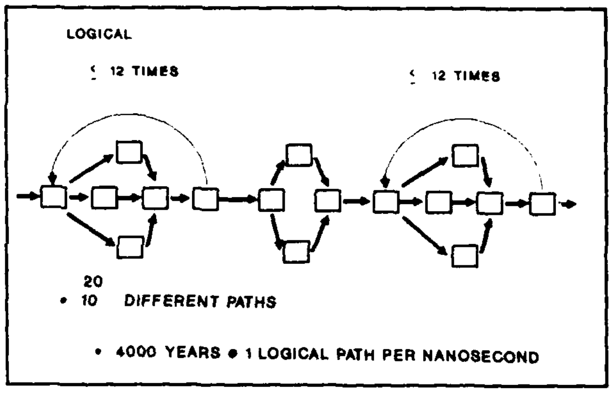


Fig. 6-10 White Box Testing


LOGICAL
≤ 12 TIMES
≤ 12 TIMES
• 10^{20} DIFFERENT PATHS
• 4000 YEARS @ 1 LOGICAL PATH PER NANOSECOND
Fig. 6-11 Software Complexity

The goal of white box testing then is also to minimize the number of errors in the delivered code and to provide reasonable assurances that the software performs according to the specification.

6.4.2.3 Top-Down/Bottom-Up Testing

In actual practice the best course of action is to develop a software test strategy that incorporates both black box and white box testing philosophy. Test procedures, therefore, should make use of both the specifications and the existing listing. But regardless of what test philosophy is used, there remains the choice of whether to perform bottom-up testing or top-down testing.

Bottom-up testing begins with the lowest level module or unit (i.e. one that does not call any other module or unit) and tests them through the use of dummy modules or units called "drivers" (Figure 6-12). These drivers are coded and used for the specific module test and then replaced when the next higher level module is coded and integrated.

Top-down testing begins by testing the highest level module first and then progressively integrating and testing lower level modules. A higher level module requiring lower level modules uses dummy modules that are called "stubs" (Figure 6-13). The stub may be as simple as sending a message such as "This stub is a replacement for the sorting module" or as complicated as performing dummy commands to simulate actual processing times.

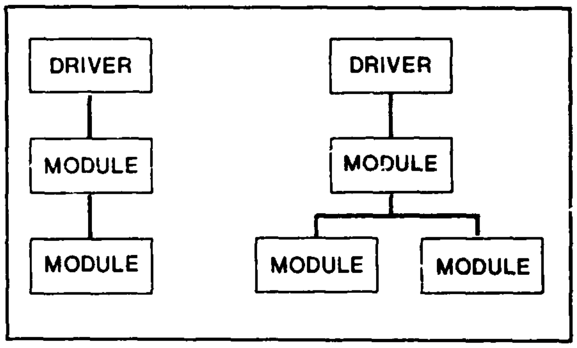


Fig. 6-12 Bottom-Up Testing

There are benefits and problems associated with both approaches. In the bottom-up approach a substantial amount of time is spent in designing, coding, and testing module drivers[1]. Since these drivers will only be used for testing, it is somewhat wasteful of a programmer's time. In addition many significant timing problems may be masked and not discovered until late in the test cycle when both schedules and resources are tight. On the plus side, if the driver was properly designed, the next higher level module can be tested with some assurances that the lower level modules are correct.

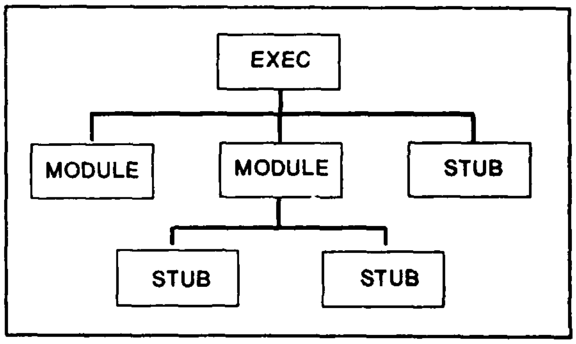


Fig. 6-13 Top-Down Testing

In the top-down approach, there is a tendency to code the easier modules first. Often complex and critical modules are not coded and tested until late in the test cycle because of the difficulty in predicting module complexity. These modules may sometimes require more time and energy than was initially planned or is currently available. On the plus side, once a higher level module is coded and tested, it is repeatedly tested as more and more stubs are replaced by fully functioning modules. This repeated testing will test the internal consistency of each module and its interface with the rest of the software. It will also reveal major timing problems very early in the test cycle while there are sufficient resources to adequately solve major problems. This evolutionary approach helps to build up maturity and confidence in the software.

In a manner analogous to black box and white box testing, the testing philosophy should incorporate both top-down and bottom-up testing. At the outset, the most complex and

[1] Software drivers should never be discarded since they will be needed during the support phase.

critical modules may be designed and coded from the bottom-up and the rest of the system can be designed and tested from the top-down. The key is to accurately determine the critical and complex modules; not always an easy task.

6.4.2.4 Software System Testing

Integration and testing of the software as a whole entity can begin once all the modules in the configuration item have been coded and individually tested by the programmer. One must remember that when the software is tested as a system, you are not trying to duplicate the results of the various lower level tests. The main purpose of the software system test is to ensure that the coded software, meets the overall performance objectives of the software requirements and system specifications. In other words, does the assembled software package meet the overall system requirements?

In order to conduct software system tests, it is necessary to use the target computer or to emulate/simulate it in the software workstation or mainframe. One must also simulate the rest of the hardware and the external environment (Figure 6-14). In addition large amounts of data may have to be generated and introduced into the software according to approved test plans and procedures. Since the simulations required are sometimes extensive, the software tester must have assurances that the simulations are correct, something that may not be all that simple. Lack of adequate simulations is a major contributor to delays in software system testing. If the program is sharing computer resources with other programs or organizations, allocation of computer time and priorities must be carefully considered. Many programs have been delayed by too many organizations vying for the same resources. Purchasing, installing, and checking out another computer system takes months and leasing time on an off-site computer is not only expensive but also slows down the turn-around time.

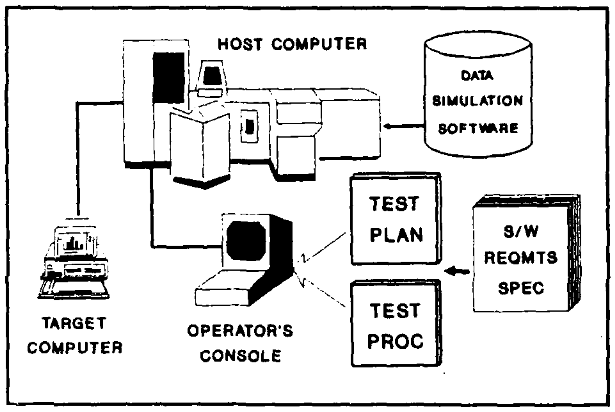


Fig. 6-14 Software System Test

6.4.3 Integration Testing

Upon completion of software system testing, systems developers embark on perhaps the most difficult, time-consuming and frustrating part of integrated testing--hot bench testing.

6.4.3.1 Hot Bench Testing

If the system software works when tested as a CSCI, the next step is to marry it to actual system hardware configured in a test setup known as a hot bench (Figure 6-5). Other common names for the hot bench are systems integration lab (SIL) or systems integration facility (SIF). The objectives are to test the software in a laboratory environment which closely approximates the actual system to be fielded. Besides the actual black boxes, the hot bench will include actual cable and wire runs, actual data busses, and any ancillary equipment required to test the system (e.g., terminals, printers).

Once you start testing the software with real hardware, you will discover that the real world is a very "noisy" environment. All of a sudden problems will be propagated throughout the hot bench that are due to external causes. For example, a large air conditioning compressor kicks in or the copying machine next door is getting extensive use and large transients are induced into your equipment. Electrical transients in digital system can create all sorts of unwelcome behavior. Sensitive instruments are affected by stray radio frequency (RF) signals and computers do funny things when the humidity climbs above a certain threshold level. Couple this with the traditional difficulty encountered in determining whether a problem is caused by the hardware or the software and one soon realizes that an inordinate amount of time is spent chasing ghosts, random malfunctions, and other elusive spirits.

Since this testing usually takes place towards the end of Full Scale Development and coincides with the initial phase of Production, test personnel are vying for resources with various other groups. This means that the available black boxes are in demand by the production people, who are attempting to build the systems, the Automatic Test Equipment (ATE) folks, who are building the automated software to test these same black boxes, and the test personnel conducting quality, environmental and stress tests.

Obtaining these black boxes may become very difficult and lack of adequate backups for failed components is a major contributor to test delays. Because of the usually high cost of these items, early planning for these assets will not solve all the availability problems. One may not be able to afford to simply buy more of the hardware required. A compromise is to share the black boxes. Test schedules are often driven, not by the complexity of the tests, but by the availability of resources. Asset management is an important aspect of any hot bench testing.

Other factors which must be considered are the facilities and the scheduling of these facilities. To perform hot bench testing the facility must have adequate power and air conditioning to handle the anticipated power and heat loads. TEMPEST requirements, as well as other security requirements, must be adhered to if the testing will generate or use classified data. All of this requires careful planning since design and development schedules may change and the facilities may not be available when they are needed. In addition, if the hot bench testing slips too much, conflicts may arise with other programs which may have planned to use the same facility space. For these reasons, it is a good idea to plan for these unexpected contingencies.

Many of these problems can be minimized if the test requirements are adequately addressed in the TEMP and the Software Test Plan.

6.4.3.2 DT&E/OT&E Testing

The next phase of testing is by far the most expensive of all. It involves performing field tests in an environment which closely approximates the environment of the anticipated threat. If flight testing is involved, one must include the test aircraft, chase aircraft, test ranges, logistics support for all the aircraft and test equipment, large data reduction systems and software and the hundreds of personnel required to conduct and coordinate the various tests (Figure 6-6). Clearly hundreds of thousands to millions of dollars are tied up in field testing major systems.

Logistic support for DT&E must also be planned. Failure rates for electronic equipment must be anticipated and provisions made for backups and repair of failed components. Repairs may be done organically for OT&E but for DT&E the repairs are usually performed by the contractor.

Scheduling of "national assets" such as the China Lake Naval Weapons Center or the Edwards Flight Test Center will have to be done years in advance when the crystal ball is pretty cloudy. Schedule uncertainties, unpredictable weather and unforseen technical problems will force the test manager to constantly generate contingency test plans to accommodate program requirements with those of other legitimate DOD users requiring the use of these same national assets.

Data reduction can be a significant bottleneck. During any major field testing activities, prodigious amounts of data can be generated. Adequate computer resources and sufficiently tested support software packages are required. It is not uncommon to wait one or two weeks for data to be reduced and presented in a format suitable for engineering analysis. If in addition to your own generated data, you are required to have a test range or some other facility record data for your test, additional bureaucratic delays can further add time to your test program.

The most important thing, however, is to realize that field testing should confirm hot bench results. If you're not ready, don't fly. If it doesn't completely work in the lab, don't waste program resources using the field as a laboratory.

6.5 TEST TOOLS

It is an ironic fact that the very industry that has automated major segments of other industries, has itself failed to automate. Thanks to computers and software, robots and automation are major players in such diverse industries as automobile manufacturing, publishing, and textiles. When it comes to software development, however, it is all virtually manual labor. It still requires a person to translate a set of requirements into a set of computer instructions. Computers that generate code directly from human languages are still far in the future.

This doesn't mean that there are no software tools that can be used during software development. Numerous tools do exist and, although they all require extensive human participation, they are very useful nevertheless. These test tools can be classified into the following categories:

Requirements Analysis Tools - Provide the capability to incrementally state system requirements and to systematically check for consistency and completeness.

Text Editors - Used to insert, delete, or manipulate portions of any text such as documentation, test data or code.

Test Generators - Allow test data input files to be developed in a systematic and standardized fashion. Sometimes the test data can be automatically generated during the coding process.

Static Analysis Tools - Perform a static analysis of a particular program to produce a listing of questionable coding practices, departures from programming and coding standards, isolated code, symbol tables, etc.

Debugging Tools - On-line tools that enable programmers to interact with their program during program execution in order to assist them in locating errors.

Environmental Simulators - Allow for the simulation of the environment of a particular system when testing in the real environment is either impractical or too expensive.

System Simulators - Simulations of other systems which interact with the system under development or simulations of major subsystems not yet available.

Data Reduction/Analysis Tools - Software and computer packages that allow large volumes of data to be analyzed and reduced to a format easily digestible by engineers and managers.

6.6 DEBUGGING

Debugging is the activity of finding and correcting a known error. It is performed after a test run has uncovered errors in a program. There are several methods for debugging: brute force, induction, deduction, backtracking, and testing [2].

The brute force method, although requiring little thought, is the most inefficient of all debugging methods. This is because it usually takes the form of (1) core dumps, (2) scattering print statements in the code or (3) using automated debugging tools which require labor intensive analysis to trace the error.

Debugging by induction requires careful thought and follows the classical inductive method of examining the data, looking for relationships, devising a hypothesis and proving the hypothesis by locating the error.

Debugging by deductive reasoning proceeds from general theories or premises about the cause of an error, eliminates and refines the premises, arrives at a conclusion and finds the error.

Backtracking involves starting at the point where the incorrect result was produced and reverse engineering the program to discover the source of the error.

Debugging by testing involves the generation of small samples of test data in an attempt to isolate the problem.

Of all programming activities, debugging is probably the most disliked of all. This is because debugging is a mentally taxing effort and quite often it is very damaging to sensitive egos that see programming errors as personal failures or indictments of their programming competence. It can be very difficult because many faults can be caused by errors in a program statement anywhere in the entire program. Developments in the state-of-practice of software engineering are minimizing these types of errors.

There are certain debugging principles that have arisen out of the collective experience of programming [2]. These are:

(a) Beware of using debugging tools. They often introduce more problems then they solve and often require a non-trivial investment of time in order to become proficient with the tool;

(b) Experience has shown that where there is one bug, there are usually more because it has been shown that errors tend to cluster;

(c) The probability of a fix being correct is never 100%. Correction of an error often introduces additional errors requiring further debugging;

(d) It is always best to change the source code and not the object code. Correcting the object code throws the source code and the object code out of sync and may produce disastrous results elsewhere in the program or later on in the execution;

6.7 PROGRAM MANAGEMENT GUIDANCE

The following suggestions are presented to provide some general guidance during the planning and execution of software test and evaluation.

(a) If an IV&V organization is going to be used, bring them on board as early as possible; preferably during the Concept Evaluation phase. Just a handful of people (2-3) performing an IV&V role can be invaluable.

(b) When evaluating contractors' proposals and Software Development Plans make sure that some organization other than the one developing the software performs the testing. The developing organization may develop test plans, but it should not test its own software. This independence can be internal or external to the contractor;

(c) As part of the evaluation process, insure that a good quality assurance organization is in place. Since software development is so labor intensive, it is important that an organization exists to impose programming standards, procedures and regimentation on the development process and to evaluate both the process and the product;

(d) Perhaps one of the most important things to remember is that failure to impose strict discipline into the software development process is the quickest way to ensure disaster. But this discipline must exist in the program office as well. The program manager must be absolutely resolute in insisting that a software developer complete an activity before proceeding to the next one. If the developer is not ready for a PDR or CDR then reschedule them. If they haven't finished software testing, then don't proceed into hot bench integration.

(e) The program office should have the eventual user, the supporting activity, and the IV&V contractor (if there is one) participate in the thorough review of the Software Test Plan. They should be critically examining whether:

- all test resources have been identified;
- all the test software tools are available, are matured and currently in use;
- the test schedule is adequate. Remember that software development is almost always on the critical path;
- the developer really understands software testing principles;
- testing is complete and has demonstrated operational performance and reliability.

(f) Seek help at the very beginning of the program don't wait until software development is in trouble because then it is too late. It doesn't hurt to have an outside organization examine the status of the software at critical intervals during the development cycle;

(g) Insist that program office personnel attend software training courses on a periodic basis, especially young software project officers. Your program will derive benefits by allowing your personnel to keep up with software practice and technology.

(h) Make absolutely sure that the program office engineering and software personnel perform a thorough review of:

- preliminary software design;
- test plans and procedures;
- traceability of requirements from the system specification all the way down to the test procedures;
- the developer's configuration management organization and procedures. On a major software development, keeping track of the various versions of software is a non-trivial task. Many a major test program has been severely impacted because the wrong version of the software has been tested.

(i) <u>Never, never, never test software that contains patches!</u> A patch is defined as a piece of machine language code that is overwritten over an existing piece of object code.

(j) Testing software with patches is nothing more than a game of Russian Roulette. You may get by once, but eventually your test program is going to blow up in your face!

(k) Insist that software errors be corrected at the source code level. Treat error correction as redevelopment by tracing all the way back from the system requirements through design and by ensuring that the documentation is updated.

(l) The only time a software patch is justified in a software test program is when it is necessary to disable a function for safety reasons or when it is impossible to conduct a particular test without the use of a patch. At any, rate once these tests are over get rid of the patch!

(m) Don't even think about fielding into an operational environment any software package containing patches!

6.8 REFERENCES

1. AFR 800-14, "Lifecycle Management of Computer Resources in Systems", 29 September 1986.

2. McCabe, Thomas J. and Gordon Schulmeyer, "The Pareto Principle Applied to Software Quality Assurance", Handbook of Software Quality Assurance, Ed. G. Gordon Schulmeyer and James I. McManus, New York, NY, Van Nostrand Reinhold Company Inc., 1987.

3. Myers, Glenford J., "The Art of Software Testing", New York, John Wiley & Sons, 1979.

CHAPTER 7
POST DEPLOYMENT SOFTWARE SUPPORT

7.1 BACKGROUND

More than two-thirds of the DOD's expenditure for software is for Post Deployment Software Support (PDSS) efforts [1]. The Electronics Industry Association (EIA) has predicted that Mission Critical Computer Software could cost the Department of Defense ten percent of it's total budget by the year 1990. Software costs are rising and will continue to rise at a proportionately greater rate than other system costs. This high cost of software will continue to rise dramatically unless corrective measures are taken to include: a greater awareness by program managers about the problems faced by software engineers; an understanding of the problems that arise in the software development process and how they can be corrected; and a change in the attitudes about software support.

7.2 PROBLEM AREAS

As discussed in Chapter 2, the growth in the use of computer technology, especially in the last twenty five years, has drastically increased the requirements for software. In the 1960's the Air Force's F-111 aircraft required less than 100 thousand software instructions. In the 1970's the Navy's P-3C aircraft required about 500 thousand software instructions. In the 1980's the B-1B is estimated to have over 1 million software instructions just for the operational flight programs and the E-3A AWACS program has about 4 million software instructions. Projections for the Space Station call for about 80 million software instructions and the future software requirements for the next generation of systems seems to be unlimited. These trends in fielded software are depicted in Figure 7-1 [1].

This increase in the amount of fielded software has obviously increased the requirement for software support services. However, this increase in software support and the lack of qualified personnel to provide these services, make software support more difficult to manage. Dr. Edith Martin in a speech presented in 1984 to the Joint Logistics Commanders Workshop entitled "The Relationship Between PDSS and Advanced Technology", estimated that the requirement for software support personnel was increasing at a rate of 12 percent per year. However, the availability of personnel qualified to do this type of work was only increasing at a rate of 4 percent. If one makes the assumption that a productivity rate of 4 percent can be achieved annually then there still exists a 4 percent shortfall in the amount of available personnel. Based on these estimates, and assuming nothing else changes, by 1990 the number of personnel required will exceed the number of personnel available by one million. There is a need to find an effective way to control both cost

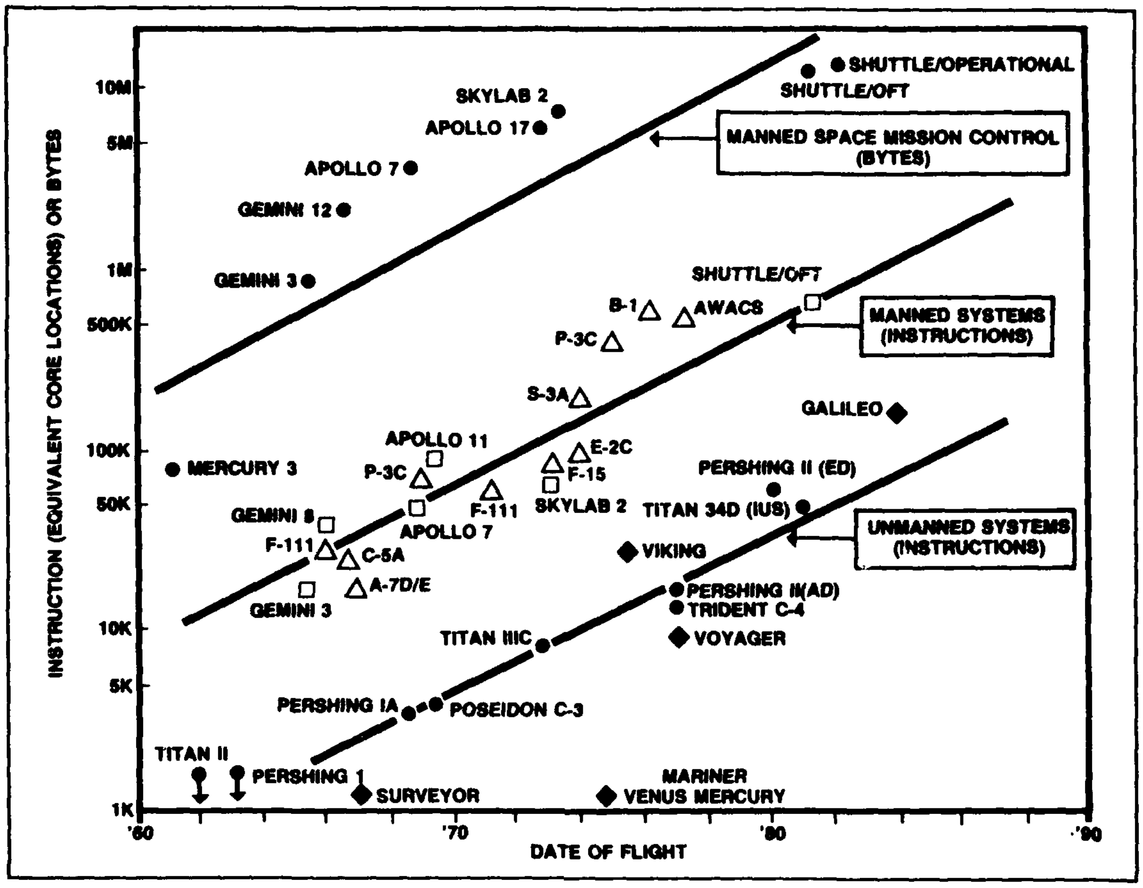


Fig. 7-1 Trends in Software

and schedule as well as to properly train, develop, and retain a cadre of software professionals for both the development of new software and the support of existing software. Possible solutions to these problems will be discussed later in this cnapter.

The EIA study discussed in Chapter 2 predicted that by 1990 software will cost the DOD about 25.6 billion dollars, and computer hardware will cost about 5.6 billion dollars. Figure 7-2 indicates that software support services will account for about seventy percent of this anticipated software cost and software development will account for the remaining thirty percent. It is all too apparent that if something is not done quickly, the government will not be able to afford the software required by modern weapon systems.

Compounding the problem of controlling cost is the fact that software support funding is fragmented. In 1984, the Industry/Government Workforce Mix Panel of the Joint Logistics Commanders (JLC) Workshop on PDSS [2] addressed the Air Force funding of system acquisition and software support. Even after deployment, hardware and software are budgeted, funded, and prioritized by separate processes and channels. Multiple procedures for budgeting and funding often are required for the same item; as opposed to separate budget and funding codes

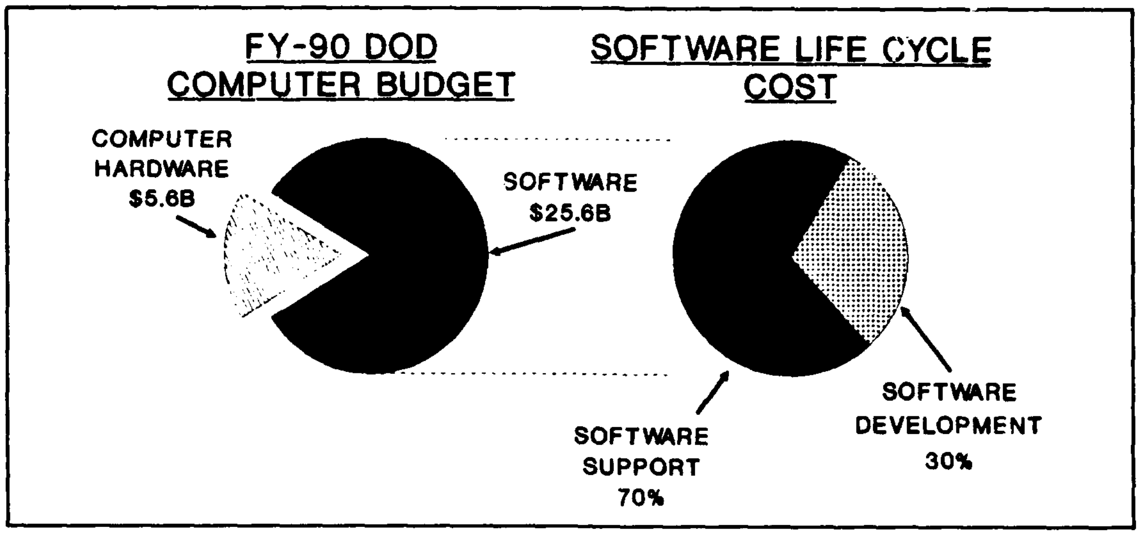


Fig. 7-2 Software Costs

for separate but related items. All of this creates confusion for the program manager or anyone else trying to get a handle on life cycle costs. Actual cost tracking can be difficult to deal with and requires careful coordination of one-year R&D software money with three-year hardware procurement money for system modifications. Further difficulties can develop when one considers that PDSS may be divided between depot and field-level activities.

The Air Force has problems in clearly defining the responsibilities of the depot and those of the field activities. This not only makes it difficult to split up the workload, but adds to the funding problem. The other services have similar problems that make the funding process difficult. If the government is to get a handle on cost, then the method used to fund the development process and PDSS must be streamlined. There should be a common budgeting procedure used by all of the services and by all levels within the government. These procedures should identify appropriations, budget programs, program elements, and specific funding codes for weapon systems using a single, simplified process for setting priorities. If this is accomplished, the government and program managers may at least get a better picture of the funding issues while affording them a way to deal with it in an intelligent manner. The way to deal with the current situation is to become aware of the difficulties and to plan in advance the transition of software support. This is done by clearly defining the roles and responsibilities of the various support organizations, including the funding, in the Computer Resources Life Cycle Management Plan (See Section 7-7).

7.3 MANAGEMENT PERCEPTIONS

One of the realities of life is that people have perceptions about software which may not be true. Most people understand that hardware is tangible, material intensive and produced by machine, while software is intangible, labor intensive, and produced by hand. They further accept that hardware deteriorates over time, is hard to change, requires preventive maintenance, and has relatively high production costs. Software is believed to have exactly the opposite characteristics. Software does improve over time with updates, requires no preventive maintenance, and has only trivial production costs; but is never easy to change correctly. From a historical perspective, these perceptions tend to cause a management problem which needs to be resolved if PDSS costs are to be brought down.

There is a widespread belief that PDSS is a trivial task and less important than software design. Unfortunately, this results in less qualified personnel being assigned to do software support rather than the more skilled technicians and managers. The feeling is that software support personnel should possess the same skill levels as the local auto mechanic while software design personnel should have the skills of automotive engineers.

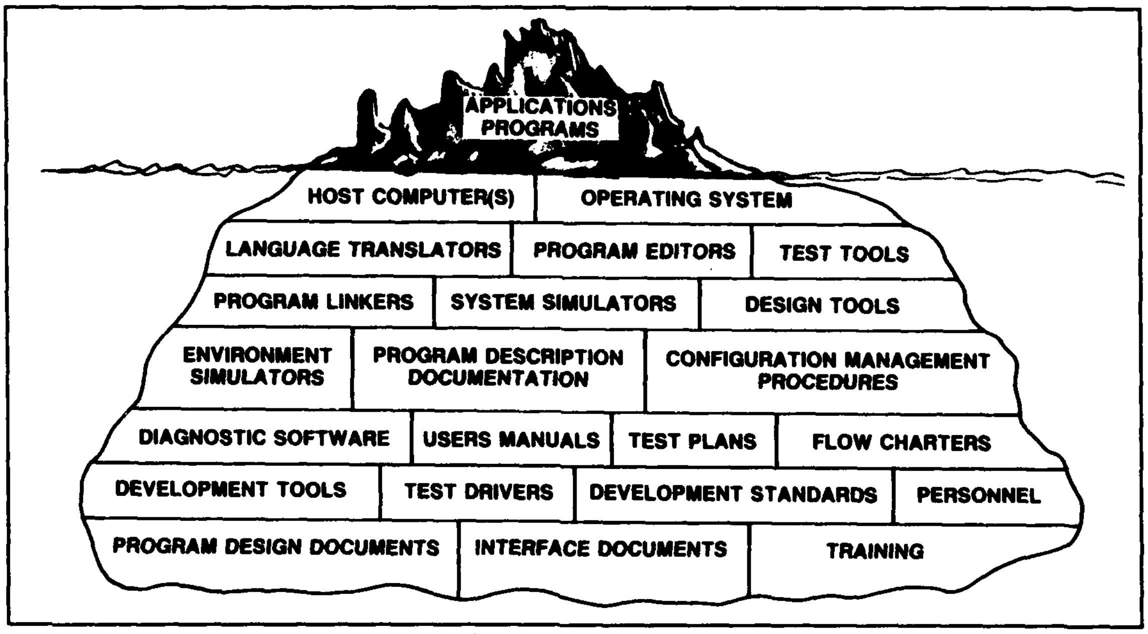


Figure 7-3 The Software Iceberg

Another misconception that plagues some program managers is a lack of understanding as to what constitutes the software products that must be supported. The software iceberg (Figure 7-3) is a graphic description of some of the various tools, documentation, etc., that form the total software package. In order for the PDSS facility to properly support the product, it must have the same basic tools and information that was available to the developer. One should also realize that many of these items can have their own icebergs. There is a tendency to cut costs in software programs by cutting out those things that are required later on in the program's life-cycle. There seems to be a lack of understanding on the part of some program managers on the value of procuring items such as simulators, editors, compilers, test tools other software tools, and documentation. Some may even feel that such items are gold plating. The fact is that without these items software cannot be properly supported.

7.4 MANAGEMENT CONCERNS

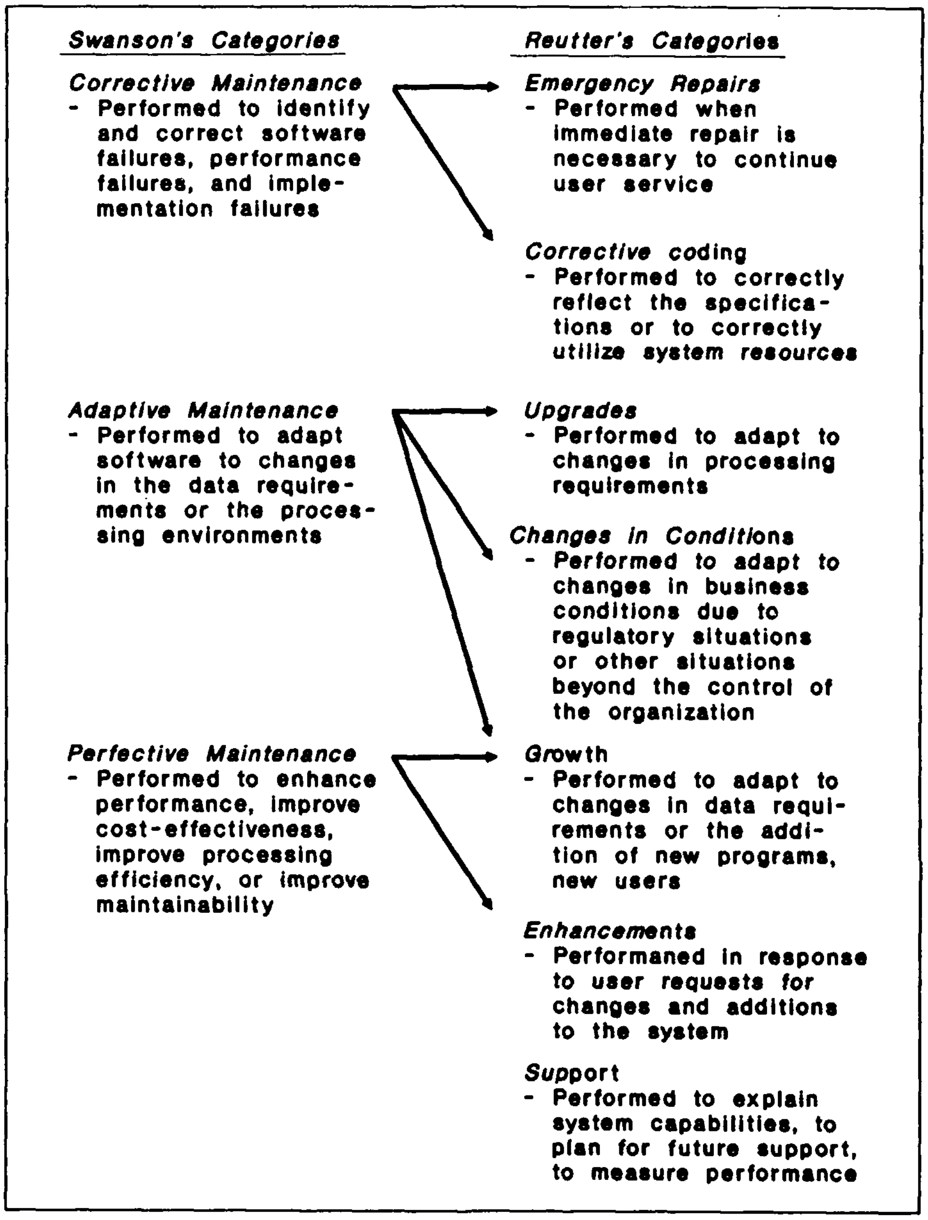
Program Managers must understand the needs of the user. They are often faced with the need to change the software because of a new requirement (e.g., a changing threat) or to correct a problem. All program managers should ensure that not only is the software developed to be supportable, but that the capability to support the software by the government or industry exists. This determination of support must include the people who will actually do the work as well as the facilities and tools necessary to accomplish this effort. Cost and schedule are obvious drivers in the decision process. However, software considerations must be put in perspective with the total weapon system. Any change to the software may have an impact on the hardware. In addition, any changes made to the software may impact other software and/or future system requirements, just like hardware changes have impact on the system.

7.5 WHAT IS PDSS?

Software does not fail in the classical sense. Hardware degrades over time as components wear out. A software problem is due to an error that has existed since its creation. When a problem caused by a component failure is found in hardware, the solution entails bringing the item back to the original configuration. In software when a problem is found and corrected, a new configuration is created. Software does not wear out like hardware; so "software maintenance" is a misnomer. The appropriate name for this effort is "software support". The Joint Logistics Commanders in 1984 decided that in order to answer the many questions about software support, a definition of Post Deployment Software Support was needed. Further, such a definition should provide a uniform basis for understanding and dealing with software support issues. The JLCs have defined PDSS as follows:

> "Post Deployment Software Support is the sum of all activities required to ensure that, during the production/deployment phase of a mission critical computer system's life, the implemented and fielded software/system continues to support its original operational mission and subsequent mission modifications and production improvement efforts."

This means that software is modified to either correct a problem or to add a capability to the software. There are other ways to look at the various software support activities



Swanson's Categories

Corrective Maintenance
- Performed to identify and correct software failures, performance failures, and implementation failures

Adaptive Maintenance
- Performed to adapt software to changes in the data requirements or the processing environments

Perfective Maintenance
- Performed to enhance performance, improve cost-effectiveness, improve processing efficiency, or improve maintainability

Reutter's Categories

Emergency Repairs
- Performed when immediate repair is necessary to continue user service

Corrective coding
- Performed to correctly reflect the specifications or to correctly utilize system resources

Upgrades
- Performed to adapt to changes in processing requirements

Changes in Conditions
- Performed to adapt to changes in business conditions due to regulatory situations or other situations beyond the control of the organization

Growth
- Performed to adapt to changes in data requirements or the addition of new programs, new users

Enhancements
- Performaned in response to user requests for changes and additions to the system

Support
- Performed to explain system capabilities, to plan for future support, to measure performance

Fig. 7-4 Types of Software Support Efforts

other than breaking down the efforts into modification and the addition of capabilities. Figure 7-4 [3] looks at two different approaches. Swanson [4] breaks down software support into three categories. The first category talks about corrective efforts, examples of which might be the correction of a problem in reading a file in a record properly, or a failure to test for all possible conditions. The second category deals with adaptive efforts which include such things as improving processing speed or adjustments to add new input or output devices. The last category includes software that is modified to make enhancements in response to new requirements, or to give the operator more flexibility. Reutter [5] refined these by breaking them into the seven categories shown in Figure 7-4. Reutter's seven categories separate support from each of Swanson's three categories. The intention of this breakout is to emphasize the communication between the user and the support activity. It also shows the importance of planning for support.

There is some disagreement among software people as to whether the above efforts should be called software support or software maintenance. The term software maintenance gives the impression that the effort involves preventing components from wearing out or breaking as in the case of hardware. As already indicated, this is not the case for software and the reason why the JLCs use the term software support. Also the term maintenance does not convey a feeling that design work is being carried out. In any case, the amount of work required to correct deficiencies in software accounts for only about twenty percent of the total software support effort [6]. This means that about eighty percent of the PDSS efforts are spent in adding new capability to the software; or at least adding things not clearly spelled out in the original requirements. This is one of the primary reasons to produce specifications and requirements which clearly reflect the user's needs.

7.6 SOFTWARE LIFE CYCLE CONSIDERATIONS

One of the major reasons for the high cost of software support is that it costs much more to correct problems after the initial design has been completed than it does to correct problems early in the development phase. Modifications require the same development activities as the original process because software support is redevelopment. It is extremely difficult to understand someone else's design, to figure out what they were trying to accomplish, to correct any problems, or to add new capabilities. This can become impossible if the documentation is poor.

Figure 7-5 indicates that while most errors are introduced during the early phases of the software life cycle, they are not usually found until the software is being validated or supported. Barry W. Boehm [1] indicates that the cost of correcting software or adding lines of code to existing software increases as the life cycle progresses. In fact, there can be as much as a hundred to one increase in costs. In a study performed in 1978 [7], the DOD found that the average cost of generating a line of code is about seventy five dollars while the average cost of modifying a line of code late in the development cycle or after software delivery is four thousand dollars. This high cost of modifying software, coupled with the fact that most errors are not uncovered until late in the development cycle, tends to significantly increase software life cycle costs.

7.7 IMPROVING THE PDSS PROCESS

There are a number of actions that can be taken to improve the PDSS process and to reduce software life cycle

	DEV $	ERRORS INTRODUCED	ERRORS FOUND	RELATIVE COST OF ERRORS
REQUIREMENTS ANALYSIS	5%	55%	18%	1.0
DESIGN	25%	30%	10%	1-1.5
CODE AND UNIT TEST	10%			
INTEGRATION AND TESTING	50%	10%	50%	1.5-5.0
VALIDATION AND DOCUMENTATION	10%			
OPERATIONS & MAINTENANCE	--	5%	22%	10-100

Figure 7-5 Software Life Cycle Considerations

costs. One of the most important is the creation of a document that spells out all of the activities that must be accomplished during the life of the software program. Within the DOD this document is called the Computer Resources Life Cycle Management Plan (CRLCMP). The CRLCMP is developed early in the acquisition cycle to ensure that all issues and resources relevant to the acquisition, testing, and support are properly accounted for. Both the Navy, with OPNAVINST 5200.28 [8] and the Air Force with AFR 800-14 [9] require that the CRLCMP be initiated by the developing activity during Concept Exploration and updated as the program progresses. The CRLCMP is approved prior to Full Scale Development. However, it should be clearly understood that the CRLCMP is a living document and should be updated whenever the software is modified. A rule of thumb is to update the CRLCMP at least annually.

The CRLCMP describes the total software support strategy. It defines the criteria for measuring progress and identifies the resources needed to develop, test, acquire, and support computer resources (e.g., facilities, personnel, hardware, software, training, funding, tools). The CRLCMP identifies the regulations and operating instructions that will be used to manage the system software. It also identifies all the organizations involved in the acquisition and support and their roles, responsibilities and relationships. The Integrated Logistics Support Plan (ILSP) is the parent document to the CRLCMP and defines the overall supportability strategy. Therefore, the CRLCMP should be closely coordinated with the ILSP.

Managing PDSS is managing change. It is therefore important to put a rigorous change control process in place during development that can transition and be effective after deployment. The program manager and/or support manager must understand why software changes are needed and what resources are needed to make appropriate economic and technical trade offs. An effective change control system allows the program manager to make these decisions.

PDSS managers must possess an understanding of their role in the PDSS process. They must understand the motivation for a software change. Is a change required to add a capability that the user needs, or to correct a deficiency? If a change is required, are the personnel necessary to implement the change available and capable of performing the task? Has the program office provided the software support personnel the tools and resources necessary to successfully make changes? When a change is to be made, will there be any disruption to current services? What is the length of time required to make these changes? What will be the impact on the integrity of the software as a whole and what are the ramifications of this change for future changes?

In 1984 the JLC report on the "Cost of Ownership" [2] concluded that program managers must have an understanding of how each PDSS activity was organized and how it functions within their own services. To this end, they provided the following descriptions of PDSS centers:

> **Army**: The Army PDSS center is a center within a DARCOM subordinate command established to support the software subsystems of all battlefield automated systems for which that command has logistics support responsibility. Each center normally supports numerous systems.
>
> **Navy**: The Navy PDSS centers' functions and staffing are provided for by the In-Service Engineering Activity assigned the life cycle system support responsibility. Note: that system may be an aircraft avionics package, a shipboard navigational system, or a shore-based Command, Control, Communications and Intelligence system.
>
> **Air Force**: The Air Force provides for a PDSS center as part of Integrated Support Facility (ISF) which is used to provide all hardware and software engineering support. This ISF is located in the engineering division or branch which supports the system program director (SPD) in an Air Logistics Center.
>
> **Marine Corps**: The Marine Corps has established a single PDSS center completely separate from hardware maintenance facilities. This center provides support for designated Marine Corps Software programs.

1. Organization Chain:

Within the services, PDSS centers are located either in a logistics chain, a Research and Development (R&D) chain or a combination of the two. The Navy has a combination chain with a single boss. The Air Force PDSS center is in the logistics chain, but receives direction from a logistics boss via the R&D chain. The Army established 11 PDSS centers located at the development commands, but funded by the readiness organizations within combined commands. Overall management of the PDSS effort is performed by DARCOM.

Coordination between R&D and logistics is always difficult. Having a single boss reduces the difficulty to some degree.

2. Development of Policy and Compliances:

Higher level headquarters establishes policy, publishes implementing instruction and ensures compliance by the PDSS centers within their individual commands.

3. How Funded:

The Air Force is Operations and Maintenance (O&M) funded unless a major rebuild is required; then the system goes back to the developer and R&D funds are used. The Navy primarily uses O&M funds, but would also send major modifications back into a R&D cycle. The Marine Corps uses R&D funds. The Army uses a plethora of funding ranging from numerous types of O&M, procurement and R&D dollars.

A standard approach to funding and a better definition of maintenance would help reduce some of these overly burdensome requisition and accounting functions.

4. Acquire Software Environment:

In all the services, the PDSS centers, in conjunction with the developer, identify support requirements. In the Navy and Air Force the acquisition manager is responsible for procuring the initial suites of equipments, and the PDSS center is responsible for updating/replacing that equipment. In the Army, no defined responsibility exists to ensure that the developer acquires the support environment, including mockups and simulators.

5. How Location Is Determined:

The Air Force locates the PDSS centers within the system program directorate (Air Logistics Center) where sustaining engineering is also located. The Navy collocates the PDSS centers with the activity responsible for in-service engineering support. The Marine Corps only has one PDSS center whose command has the logistics responsibility for the system or has computer resources. In the event that the system is a command and control system, the PDSS center is collocated with the battlefield functional area school.

6. How System is Learned:

The Army and Air Force PDSS centers become involved at the beginning of the development cycle through either participation in the developmental process or by being the IV&V agency. The Navy may follow the same procedure, depending on when the PDSS center is designated. The Marine Corps PDSS center has previously been involved as part of the development responsibility and have replaced it with more of an IV&V type role.

The involvement of the PDSS centers throughout the development cycle, commencing with Milestone I, is considered to be critical to the successful execution of performing PDSS work.

7. Use Of the PDSS Center For IV&V:

There is currently no stated requirement to perform IV&V in any of the services, and there is a wide variance of how the services accomplish IV&V. In those programs where there is a requirement for IV&V, the PDSS center is a logical choice and should be used to the maximum extent practical.

8. Software Configuration Control:

The PDSS centers of all services perform configuration control, but none are tasked with performing configuration management.

9. Type of Changes:

There are basically three types of changes: those brought about by latent defects; those brought about by enhancement requests from the users; and major product improvements. In all the services, the PDSS centers perform the changes brought about by the first two. Changes resulting from major product improvements are usually accomplished by a contractor, with the PDSS center providing background information and support.

10. Evaluation Of Complaint:

The Army maintenance directorate sends logistic support representatives to the user activity to investigate complaints. Once the problem is identified and verified to exist, the maintenance directorate notifies the PDSS centers who then attempt to duplicate the problem. In the other services, the PDSS centers receive the trouble report directly from the user and attempt to duplicate the problem. Responses back to the complaining user vary from periodic status reports about the complaint to not providing any follow-up information.

11. Develop Software Engineering Change Solution:

In all the services, once the problem has been identified, the cause is explored by software engineers within the PDSS center. Solutions are developed and testing is conducted. This testing is to ensure that the original problem has in fact been solved and that additional problems have not been created.

12. Integration Testing:

In all the services, integration testing is performed when the PDSS center has completed system testing. With the exception of the Navy, testing is always performed on the actual equipment being integrated. In the Navy, the size of the integration problem often prevents the PDSS center from conducting the integration testing in a totally realistic environment. The software which has been modified is run on actual system hardware, but those systems with which it communicates (i.e., integrated) may be simulated. The limitations on integration testing of Navy shipboard combat systems due to the availability of actual systems are recognized and organic integration facilities have been or are being established. These facilities are outside the PDSS centers' responsibility and control, but are available to the PDSS center for use.

13. Interoperability Testing:

All the service PDSS centers conduct interoperability testing to ensure that changes made to correct problems will in no way interfere with the capabilities of the system to communicate with other systems.

14. *Documentation Update:*

All the service PDSS centers update documentation for every change made. The major problem being experienced is the inadequacy of standards and resulting initial documentation. Standards must be published that meet the needs of all services and contracts must be written to require documentation in accordance with these standards. Waivers must not be permitted.

15. Distribution Of Software Corrections To Users:

For systems where there is limited distribution, changes are hand-delivered and accompanied by sufficient instruction to allow the user to execute a smooth transition. In those cases where there is a large number of equipment in which changes must be installed, they are supplied with a written instruction package through a distribution process.

7.8 PROGRAM MANAGEMENT GUIDANCE.

By changing the traditional prejudices toward software support, by following sound engineering practices, and through the use of good management tools and plans, program managers can begin to deal with the software life cycle support problems. The following represents a set of guidelines which can be followed by the program manager and can ultimately lead to more cost effective software support:

(a) The program manager must ensure that sound software engineering design techniques are used during the development process where the majority of all errors are induced and that the software is designed for supportability;

(b) The CRLCMP must be developed early in the program life cycle (Concept Exploration) and periodically updated and adhered to;

(c) Personnel and productivity are important issues. It is incumbent upon program managers to acquire and retain a qualified and stable workforce. This may require the program manager to develop a continuous training program for software personnel. In order to get more out of the personnel, program managers must provide the supporting activity modern tools and facilities;

(d) Documentation and support software tend to be cut out of programs early in the development process in order to save money. This causes severe problems later on in the program. It is important that all the tools necessary to provide software support are delivered to the support facility;

(e) The software baseline must be controlled throughout its life cycle by using good configuration management techniques;

(f) It is important that the program manager involve the PDSS organization early in the development. The personnel at these facilities have a wealth of knowledge about what is required to support a program and about the associated problems. This provides the program manager with a real time lessons learned;

(g) It is imperative that the program manager appropriately plan, budgets, and fund the PDSS effort.

7.8 REFERENCES

1. Boehm, Barry W., Software Engineering Economics, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

2. Final Report of the Joint Logistics Commanders Workshop on Post Deployment Software Support for Mission Critical Computer Resources, Volume II - Workshop Proceedings, June 1984.

3. Martin, James and Carma McClure, Software Maintenance - The Problem and its Solutions, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

4. Swanson, E., "The Dimension of Maintenance", 2nd International Conference on Software Engineering, Proceedings (San Francisco), October 13-15, 1976, pp. 492-497.

5. Reutter, John, "Maintenance Is a Management Problem and a Programmer's Opportunity", AFIPS Conference Proceedings on 1981 National Computer Conference (Chicago), Vol. 50, May 4-7, 1981, pp.343-347.

6. Lientz, B., and E. Swanson, Software Maintenance Management, Reading, MA: Addison-Wesley Publishing Co., Inc.,1980, pp. 151-157.

7. De Rosel, B., and T. Nyman, "The Software Life Cycle - A Management and Technological Challenge in the Department of Defense", IEEE Transactions on Software Engineering, Vol. SE-4, No. 4 , July 1978, pp. 309-318.

8. OPNAVINSTR 5200.28, "Life Cycle Management of Mission Critical Computer Resources", 25 September 1986.

9. AF Regulation 800-14, "Life Cycle Management of Computer Resources in System", 29 September 1986.

CHAPTER 8
PLANNING FOR COMPUTER SOFTWARE

8.1 INTRODUCTION

This chapter will discuss the software planning activities that must be conducted during the various phases of software acquisition. The success of any project is greatly determined by how much care and time is put into the planning process. Software development and acquisition is no different!

Planning for computer software begins during the Concept Exploration (CE) Phase and intensifies during the Demonstration and Validation (D/V) Phase. By the time the program enters the Full Scale Development (FSD) Phase, most of the necessary software planning documents should have been generated.

8.2 PLANS AND DOCUMENTATION

The major plans and documents generated by the program office are the Program Management Plan (PMP), the Test and Evaluation Master Plan (TEMP), the Integrated Logistics Support Plan (ILSP), the Computer Resources Lifecycle Management Plan (CRLCMP), and the System/Segment Specification (SSS).

8.2.1 Program Management Plan (PMP)

The purpose of the PMP is to guide all program office personnel toward a common goal. The PMP provides essential information on the overall program strategy and goals. It can be viewed as the game plan for the program office and includes a schedule of the major events leading to the initial operating capability (IOC).

The PMP should be updated and kept current, particularly with respect to the schedules. It should be mandatory reading for every newcomer into the organization. This is especially true in software development since software is usually in the critical path. A detailed outline of the PMP is given in Appendix C.

8.2.2 Test and Evaluation Master Plan (TEMP)

The TEMP is the basic planning document for all test and evaluation (T&E) related to a particular system acquisition. It is used by the Office of the Secretary of Defense (OSD) and all DOD components in planning, reviewing, and approving T&E. The TEMP provides the basis for all other detailed T&E planning documents.

The TEMP should address the effects of human performance on the weapon system's operational effectiveness and the ability of the system to meet performance standards, including reliability and maintainability.

The TEMP should address the system's critical technical performance thresholds and their relationship to the system's required operational characteristics. It should clearly outline the planned T&E process through which the test objectives will be met. It should also describe the physical hardware tests and any analysis required to provide data not gained from actual testing.

The TEMP should clearly describe the required T&E activities along with all the necessary resources to fully test the operational suitability of the weapons system.

The TEMP is intended to be a living document that addresses the changing critical issues affecting any acquisition program. Major changes in program requirements, schedule, or funding usually result in a change to the test program. To ensure that T&E requirements are current, the TEMP shall be reviewed for updating on at least an annual basis until all significant testing is complete. The update shall reflect the T&E program changes due to test results, changes in the scope or schedule of the T&E program, changes in the required characteristics, changes in the reassessment of test resource provisions and limitations, or any changes deemed necessary by the OSD. A detailed outline of the TEMP is found in Appendix D.

8.2.3 Integrated Logistics Support Plan (ILSP)

The ILSP describes and documents the integrated logistics support program. It is the principal logistics document for an acquisition program and serves as a source document for summary and consolidated information required in other program management documents. It is summarized in the System Concept Paper (SCP) and the Decision Coordination Paper (DCP).

The purpose of the ILSP is to:

(a) Provide a complete plan for support of the fielded system;

(b) Provide details of the ILS program and its relationship with overall program management;

(c) Provide decision making bodies with the necessary information on ILS aspects for making sound decisions on further development and production of the basic system;

(d) Provide the basis for the preparation of the ILS sections of the procurement package, e.g., statement of work (SOW), specification, and source selection and evaluation criteria;

(e) Describe how readiness and sustainability will be achieved.

The ILSP is initially a section of the Program Management Plan (PMP) but, early in the Demonstration and Validation Phase, it is removed from the PMP and becomes a stand-alone document. At a minimum, the ILSP is updated annually. A detailed outline of the ILSP is found in Appendix E.

8.2.4 Computer Resources Life Cycle Management Plan (CRLCMP)

The CRLCMP is the primary planning document for computer resources throughout the system life cycle. It complements the ILSP. The purpose of the CRLCMP is to:

(a) Document the software support concept and the resources needed to achieve the support posture;

(b) Document the computer resources development strategy;

(c) Identify the applicable directives (regulations, operating instructions, technical orders, etc.) for managing computer resources in the system;

(d) Define any changes or new directives needed for the operation or support of computer resources;

(e) Define the scope of independent verification and validation (IV&V) efforts.

Development of the CRLCMP is initiated during the Concept Exploration phase. The CRLCMP is coordinated with the user and supporting organizations before release of the Full Scale Development solicitation.

During the Production and Deployment phases, the CRLCMP is updated, as required, to reflect significant changes in the system or its support environment. When updating the CRLCMP, sections that refer to accomplished events should be reworded as historical notes or deleted. After the system is transitioned to the user the software support activity will assume responsibility for the CRLCMP. A detailed outline of the CRLCMP is found in Appendix F.

8.3 ENGINEERING STUDIES

Systems engineering studies are based on the concept of a hierarchy of requirements starting with system level requirements and ending with detailed engineering specifications and data. System definition proceeds by refining each level of requirement into subordinate requirements until the entire system is described. Computer resources are considered as an integral part of the system and are subject to tradeoff and optimization studies. Systems engineering studies will normally include:

Requirements Definition. Requirements definitions begins with a preliminary allocation of requirements to either hardware or software. The requirements for each software configuration item are documented in a Software Requirements Specification (SRS). The SRS is authenticated at the Software Specification Review (SSR) which is normally held during the FSD phase.

Interface Definition. The Computer Resources Working Group (CRWG), in conjunction with the Interface Control Working Group, addresses system and subsystem interface requirements that may affect computer resources. The requirements for interfaces between Computer Software Configuration Items (CSCIs) and other system configuration items are documented in one or more Interface Requirements Specifications (IRS) that are also authenticated at the SSR. The SRS and the IRS form the CSCI allocated baseline.

Tradeoff and Optimization. Tradeoff and optimization studies should consider such issues as:

(a) Tradeoffs between computer software and computer hardware;

(b) Required computer processor architectural features such as memory size, processor speed, input and output capacity, and spare capacity;

(c) Use of standard equipment, higher order languages, instruction set architectures, and interfaces;

(d) Alternate approaches for meeting system security requirements;

(e) Improved supportability versus improved performance;

(f) Use of existing government resources or commercial off-the-shelf resources versus new development.

Feasibility Studies. These studies determine the feasibility of alternative allocations of system requirements to computer resources and the derivation of data for formulating budgets and schedules.

Risk Analysis. Using Table 8-1, the program office can identify the major risks to the software development effort. Plans for managing these risks are incorporated into the system level risk management plan or the CRLCMP.

CAUSE	ACTION
Lack of adequate definition of computer resource functional, interface, support, or performance requirements prior to structuring the program. Poorly defined, complex, or unstable intra- and inter-system interfaces, including human interfaces. Lack of stability in computer resource requirements during development	System engineering techniques such as functional analyses, simulation, mathematical modeling, correctness proofs, and tradeoff analyses. Incremental development strategies which tackle large, complex, and poorly understood requirements in smaller, more manageable parts.
Lack of government visibility into the contractor's software development effort.	Rigorous application of traditional cost, schedule, and performance tracking techniques with careful attention to earned value progress against measureable milestones. Since these techniques are almost always driven by the WBS, visibility of critical and high risk computer resources is a primary criterion for determining the appropriate level within the WBS for these components of the system. Use of a risk traking system to collect data on the status of identified high risk items. The output of this system should be a standard part of periodic reviews. Use of independent verification and validation.
Performance requirements that push the state of the art.	Prototyping or duplicate development of key algorithms, concepts, and components.
Inaccurate, poorly defined, or nonexisten cost and schedule estimates for computer resource development.	Multi-source cost and schedule estimates using a variety of estimating techniques and models. Avoid basing all estimates on "lines of code" estimates derived from a single source.
Inadequate developer and acquisition manager capability or capacity for software development. Inadequate, immature, or poorly integrated software development tools (e.g. compilers, linkers, loaders) & programming support environment	Reviews of offeror sites to assess capability and capacity of development personnel, management structure and procedures, and facilities.
Lack of adequate spare computer hardware capacity (e.g. processor speed, memory, input/output, and secondary storage.	Early planning for spare capacity during development and support phases of the lifecycle; periodic reviews of capacity allocation, and projection of requirements trends.
Undefined or poorly defined software support concepts.	Rigorous adherence to the separation of mission software and system software into separate CSCIs.

Table 8-1. Common Risks and Possible Corrective Actions

Software Support Studies. Software support studies are conducted to refine the system support concept and to allocate software support requirements. These studies determine how

operational system software will be identified. Two potential methods are self-identification of executing software and identification plates affixed to the outside of the computer.

8.4 COMPUTER RESOURCES WORKING GROUP (CRWG)

A CRWG should be established as early as possible during the CE Phase but no later than Milestone I. For modification programs, and those acquisitions closely related to ongoing programs, an existing CRWG may be used. The CRWG's role is to actively participate, in an advisory capacity, in all aspects of the program involving computer resources. This includes program management reviews, source selection evaluation boards, design reviews, and audits.

The CRWG is formally chartered by the program manager and should coordinate its activities with the operational user, the supporting organization, the interface control working group, and any other organization with an active interest in the program. At a minimum the CRWG will:

(a) Advise the program manager in all areas relating to the acquisition and support of computer resources;

(b) Generate the initial CRLCMP and update it as the program progresses;

(c) Select a software support concept and document it in the CRLCMP;

(d) Monitor compliance of the program with computer resources policy, plans, procedures, and standards;

(e) Insure that software testing is adequately addressed in the Test and Evaluation Master Plan (TEMP) and monitor compliance with this document as the program matures;

(f) Identify and prioritize the required software quality factors such as interoperability, portability, flexibility, useability, reusability, maintainability, integrity, reliability, correctness, testability, and efficiency;

(g) Define the appropriate scope of the IV&V effort and develop a recommended approach using contractor personnel or another government organization;

(i) Evaluate the use of standard equipment, higher order languages, instruction set architectures, and interfaces;

(j) Evaluate the need for development of software tools and recommend an approach to their development.

8.5 CONTRACTUAL CONSIDERATIONS

A recurring activity conducted by the program office is that of selecting one or more contractors and putting them on contract. Although the intensity of the activity and the number of contractors to be evaluated may differ, the process of selecting a contractor(s) is the same regardless of the phase of development. The three major activities performed are: generating a source selection plan, generating a request for proposal package, and conducting the source selection process.

8.5.1 Source Selection Plan (SSP)

An outline of a typical SSP is given in Appendix G. The SSP is a key document for initiating and conducting a source selection. As shown in Appendix G, the SSP should address mission critical computer resources. It is prepared by the program office and must reflect applicable Program Management Directive (PMD) guidance or direction. The SSP is a plan for organizing and conducting the evaluation and analysis of proposals and a roadmap for the selection of a source or sources.

The SSP must be submitted sufficiently in advance of the planned acquisition action to facilitate review and approval by the Source Selection Authority (SSA) and early establishment of the Source Selection Advisory Council (SSAC) and the Source Selection Evaluation Board (SSEB).

8.5.2 Request for Proposal Package (RFP)

The RFP package must be sufficiently detailed to allow responding offerors to adequately address system requirements and to provide other information necessary for evaluation and award. The RFP package typically consists of a requirements specification(s), instructions to offerors, proposal evaluation criteria, a statement of work, a work breakdown structure, requirements for deliverable items, and special contract provisions. Supporting information that expands on the system operations and support concepts, including the CRLCMP, will be attached to the RFP package.

Even though the RFP is prepared by the program office, it is good practice to solicit inputs from the using and supporting organizations. They usually provide valuable insight into the operational and support environment.

8.5.2.1 Requirements Specification(s)

The requirements specification(s) included in the RFP is dependent on the phase of the system development being undertaken. If one is contracting for the Concept Exploration phase, then the specification would be an overall system specification. For the Demonstration and Validation phase, the specification would be the System/Segment Specification and for the Full Scale Development phase, the specifications would be the Software Requirements Specification and the Interface Requirements Specification. Appendix H lists the various Data Item Descriptions (DIDs) called out by DOD-STD-2167A which govern the format and content of the required specifications.

8.5.2.2 Instructions to Offerors

In addition to specifying proposal form and content, the instructions to offerors must require submission of such documents as a Software Development Plan, a Configuration Management Plan, and a Software Quality Program Plan as part of the proposal. These plans must include the offeror's software development and management concepts and the procedures for controlling and assessing progress during the development process. Appendix H lists the various DIDs which describe the format and content of these documents.

The instructions to offerors is the mechanism for ensuring offerors address critical software issues such as:

(a) The methodology used to perform software sizing and cost estimating and the approach to be followed during software development;

(b) The rationale used for the computer resources timing and sizing estimates;

(c) A description of any teaming and subcontractor arrangements;

(d) The skill levels required for computer resources development and their availability within the corporate structure;

(e) The method to be used for risk control;

(f) Any planned use of firmware;

(g) Any plans for reusing or modifying existing software;

(h) A clear definition of all assumptions used during proposal preparation;

(i) Plans for the development of prototype software;

(j) Plans and procedures for generating and using software metrics.

8.5.2.3 Proposal Evaluation Criteria

The evaluation criteria must cover all of the requirements within the RFP. This includes computer resource development and management activities and the offeror's software management plans described in the Software Development Plan and other applicable documents. The criteria in the RFP should be listed in relative order of importance. The evaluation criteria must include the availability of software, documentation, and the rights necessary to meet life cycle needs and the compatibility of the proposed design with the support concept defined in the CRLCMP. This will ensure that the design is modifiable and that proposed support resources and methods are adequate. When the processing of sensitive or classified information is involved, the program office must ensure that computer security is also included in the evaluation criteria.

8.5.2.4 Statement of Work (SOW)

The SOW will identify the applicable program management, development, test, training, installation and support tasks that are to be performed under the contract. More specifically, the SOW will:

(a) Identify clear and concise statements of specific task;

(b) Address the planned use of an Independent Verification and Validation contractor and the type and amount of support expected from the software development contractor;

(c) Tailor all contractually required standards and specifications to the needs of the program;

(d) Address the planned use of government provided operational and environmental simulators, support equipment, or other software programs (e.g., compilers);

(e) Require comprehensive layout of program schedules to include reviews, technical interchange meetings, audits, and testing;

(f) Address the requirements for prototype software development;

(g) Address the requirements for generating and using software metrics data.

8.5.2.5 Work Breakdown Structure

A preliminary work breakdown structure (WBS) may be included in the RFP package. The contractor will be expected to develop their own WBS containing additional levels of detail.

8.5.2.6 Deliverable Items

Deliverable computer hardware and software, including support and test software, will be specified as contract line items (CLINs) in the schedule of the contract. The CLINs should specifically call out deliveries of such items as operational flight programs (OFPs), test program sets (TPSs), simulation software, and incremental deliveries of various versions of all of these. Documentation requirements will be identified in the Contract Data Requirements List (CDRL), and software media delivery requirements will be specified in the Software Requirements Specification which will be listed in the CDRL. For software, deliverable items will include complete source code in a form suitable for compilation or assembly and the complete object code in a form suitable for loading and executing in either operational or support computers. The CDRL will include the documentation needed for developing, testing, operating, and supporting the system and for training personnel. Appendix H lists the various DIDs for this additional documentation. If all the documentation needs cannot be identified before contract award, the CDRL will include a report that will identify data items needed to satisfy the system support and operational concepts. A Data Accession List should be used to identify the contractor's informal documentation to be made available for government review.

8.5.2.7 Special Contract Provisions

Special provisions will be incorporated into the contract to insure the government's right to computer software and to provide adequate protection whenever commercial off-the-shelf software is used.

<u>Proprietary Software</u> - The RFP shall require the offeror to identify and cost all proprietary computer software or equipment, associated documentation, and support items required to be delivered, or subject to order, under the contract. The offeror must also identify any other software which may have limited or restricted rights.

<u>Commercial Off-the-shelf Software</u> - Procedures must be developed and incorporated into the contract to ensure that the contractor reviews all subcontractor or vendor products and that all commercial hardware and software in the system is maintained to the correct configuration level. The contractor must be made responsible for maintaining engineering compatibility between all system hardware and software, including the incorporation of newly released versions of software. Operating system software falls into this category.

8.5.3 Source Selection Process

The principal objective of the source selection process is to select the source whose proposal has the highest degree of credibility and whose performance can be expected to best meet the government's requirements at an affordable cost. The process must provide an impartial, equitable, and comprehensive evaluation of the competitors' proposals and related capabilities. The process should be accomplished with minimum complexity and maximum efficiency and effectiveness. It should be structured to properly balance technical, financial, and economic considerations consistent with the phase of the acquisition, program requirements, and business and legal constraints. A typical source selection process is depicted in Figure 8-1.

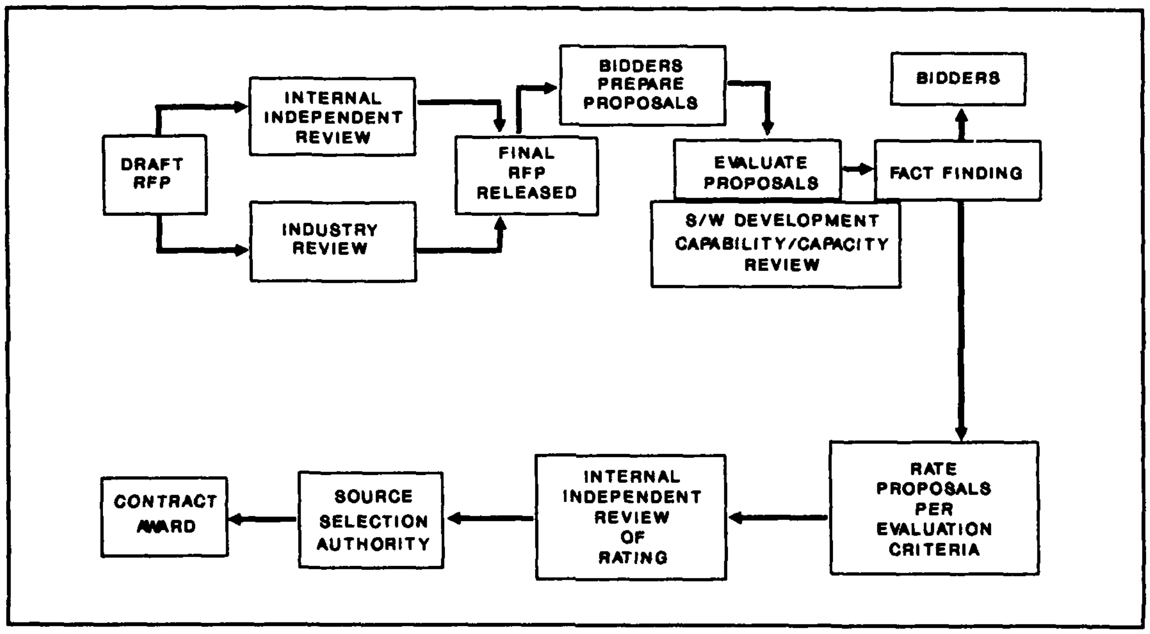

DRAFT RFP
INTERNAL INDEPENDENT REVIEW
INDUSTRY REVIEW
FINAL RFP RELEASED
BIDDERS PREPARE PROPOSALS
EVALUATE PROPOSALS
S/W DEVELOPMENT CAPABILITY/CAPACITY REVIEW
FACT FINDING
BIDDERS
RATE PROPOSALS PER EVALUATION CRITERIA
INTERNAL INDEPENDENT REVIEW OF RATING
SOURCE SELECTION AUTHORITY
CONTRACT AWARD

Fig. 8-1 Source Selection Process

8.5.3.1 Draft RFP

If it is at all possible, the process should begin with a draft RFP which consists of preliminary versions of a statement of work, a specification, schedules, a contract data requirements list (CDRL), and evaluation criteria. Unfortunately, time and resources do not always permit a program office to generate and circulate a draft RFP.

By circulating a draft RFP among the various internal government organizations such as contracts, legal, and other organizations that normally coordinate on a procurement package, the RFP can be evaluated for consistency and content. These organizations can provide valuable information and comments which help to ensure that the final RFP doesn't become bogged down because of major shortcomings.

Industry review can also be very helpful. They may provide constructive information on the potential technical, schedule, and cost risks associated with the intended procurement. For example, they may provide alternative approaches to high risk areas and indicate those areas which are ambiguous, contradictory, or likely to be major cost drivers.

All industry comments, however, should be carefully examined to separate fact from marketing information. Some companies may not be able to resist the temptation to suggest changes or additions to the RFP which could enhance their competitive posture. In spite of this danger, industry comments can be tremendously useful.

Once the comments have been received from both industry and internal organizations, a final RFP can be prepared and released. While the bidders are preparing their proposals, the program office will be finalizing the source selection organization and the proposal evaluation criteria. Although the general evaluation criteria has already been released with the RFP, specific sub-criteria and factors may still require fine tuning.

8.5.3.2 Populating the Source Selection Organization

One of the constant problems faced by program managers is finding qualified software individuals willing and able to be away from their jobs for an extended period of time. Source selections can take anywhere from three to six months to complete. Since quality software individuals are already in short supply, a program manager may have to resort to innovative means for acquiring them. If one or two experienced software people are already on the staff, then the problem becomes more manageable. The task of finding additional knowledgeable but less experienced software people is a bit easier.

If a program manager doesn't have in-house expertise, the problem is more difficult. The first place to look for experienced software personnel is another program office that may be willing to release individual(s) for the duration of the source selection. Another source could be the various laboratories within the command or service. A sister service may be able to provide temporary software expertise. There are also federally established, not-for-profit organizations such as the Aerospace Corp., the Mitre Corp., and the Software Engineering Institute whose employees routinely participate in government source selections. Other corporations exist which may be able to participate in a source selection. These corporations are primarily analysis oriented and normally do not produce or develop hardware for the commercial market. The Rand Corp. and the Charles Draper Laboratory fall into this category. The lead software individual, however, must be a government employee.

8.5.3.3 Evaluation Process

The evaluation process itself is well defined and regulated by the various DOD and service peculiar regulations and procedures. Figure 8-2 depicts the typical evaluation process which begins with the receipt of cost, technical and management proposal volumes, as a minimum. The Source Selection Evaluation Board (SSEB) is comprised of the

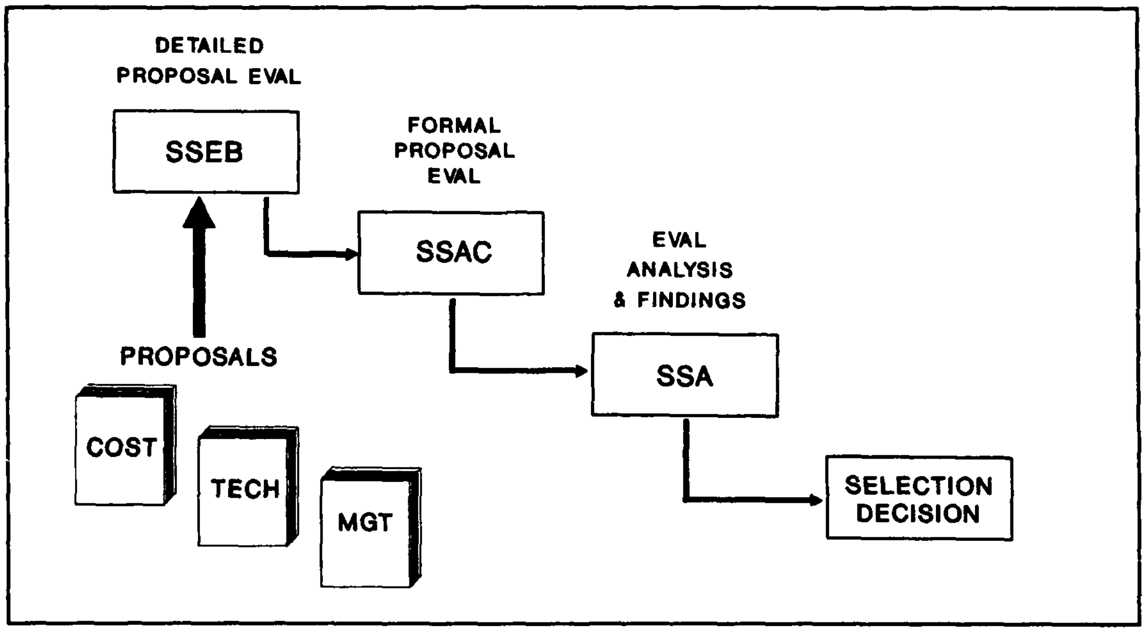


Fig. 8-2 Evaluation Process

functional area experts who actually perform the detailed evaluation. Since this is a time-consuming but critical step, it is important that the lead individual expedite the process. Endless discussions over trivial or irrelevant points cannot be tolerated. Even when serious topics are the source of major disagreements among the evaluators, the lead software individual must force the people involved to come to a reasonable and timely consensus. Minority opinions and views should be aired and, at the discretion of the lead software person, documented and presented to the Source Selection Advisory Council (SSAC) for resolution. By the same token, an individual's infatuation with a particular technology should not be allowed to unduly bias the proposal ratings. For example, if an individual is convinced that relational data bases is the only solution to a particular problem, the lead software person must ensure that this individual has given the other approaches a fair evaluation.

Once the SSEB completes its evaluation, the results are presented to the SSAC. The SSAC, which is composed of senior government personnel, then evaluates the analysis and findings of the SSEB and presents the results to the Source Selection Authority (SSA). The SSA is the official designated to direct the source selection and to make the final source selection decision.

8.5.3.4 Evaluating Offeror's Proposal

Since every program has unique requirements, it is beyond the scope of this guidebook to provide specific information on what is important in a software source selection. In general, however, the software evaluation assesses the technical adequacy of the proposed computer system architecture to satisfy the weapon system requirements. Items that are evaluated are:

(a) The throughput and memory capability of the proposed computers;

(b) Future vendor support for commercially supplied items such as tape drives, disk drives, controllers, etc.;

(c) Computer resources interfaces to the rest of the system architecture and human operators;

(d) Adequacy of the operating system or software executive;

(e) Availability, currency, and usage of software development tools;

(f) Organic supportability of computer hardware and software;

(g) The offeror's Software Development Plan and software development standards and procedures;

(h) The offeror's software development capability and capacity.

It is important to emphasize the need to perform an integrated, comprehensive evaluation of the offerors' total proposal. This usually means that technical evaluators must also review the management and cost proposals. The cost evaluators normally concentrate on accounting and costing consistency and completeness. They are not qualified to pass judgment on whether a proposed number of manhours is sufficient for a particular analysis or effort. Only the technical evaluators can make this assessment. Likewise the technical evaluators can also assess whether a particular management organization or procedure is consistent with the technical effort proposed. One is not advocating that all technical evaluators review cost and management data. What is being proposed is that one or two key individuals from the technical panel make a top-level review of cost and management data for realism and consistency.

8.5.3.5 Software Development Capability/Capacity Review

Figure 8-1 shows a software development capability and capacity review (SDCCR) occurring as part of the source selection process. The SDCCR has been successfully used at the Aeronautical Systems Division of the Air Force Systems Command at Wright-Patterson AFB. It is described in ASD Pamphlet 800-5, "Software Development Capability/Capacity Review" [1].

<u>Purpose</u> - The SDCCR is intended to review and assess an offeror's specific capability and capacity to develop the software required on a particular weapon system program as defined in the RFP. This review process is designed to be incorporated as an integral part of the FSD source selection process. The review process accomplishes three related objectives. First, the acquisition management team gains an understanding of the offeror's software development methods and tools. Second, the capability and capacity of the offeror to develop the required software in a disciplined software development process is determined. Third, the review process elicits a contractual commitment by the offeror to implement the methods, tools, practices and procedures which form the discipline and structure for this software development process [1].

<u>Process Summary</u> - The SDCCR process is accomplished during the FSD RFP preparation and source selection phase. The RFP includes the requirement that the offeror team provides specific information describing their software development methods, and include examples of how the methods have been applied on past or on-going programs. The SDCCR source selection team reviews this information and then conducts an in-plant review with the offeror's team. This review is based on a specific set of SDCCR questions which are provided with the RFP to the offerors and are found in Attachment 4 of ASD Pamphlet 800-5. Following this one to two day in-plant review, the offeror's capability and capacity to develop the software required on the program is assessed and evaluated using the predefined RFP standards. This evaluation becomes an integral part of the program source selection and forms part of the basis for the award. It is highly recommended that offerors conduct this review with their subcontractors prior to the government's in-plant review [1].

<u>Review Areas and Factors</u> - The SDCCR is usually organized into five major areas: management approach, management tools, development process, personnel resources, and Ada technology. These areas are in turn organized into factors as shown in Table 8-2. Other factors may be added to this review as a function of unique program requirements.

<u>Team Composition</u> - The SDCCR is performed by the source selection team. This approach is fundamental to achieving the multiple objectives of the SDCCR. The usual team composition is as follows:

MANAGEMENT APPROACH
- Management Organization
- Software Management System
- Software Configuration Management
- Software System Organization and Structure
- Software Subcontracting
- Software Planning
- Software Quality/Product Assurance
- Contract Control Methods

MANAGEMENT TOOLS
- Internal Management Standards and Tools
- Software Size, Manpower, Schedule and Cost Estimating
- Contract Work Breakdown Structure (CWBS)
- Software Work Definition
- Schedule Definition and Statusing
- Software Cost Performance Reporting System

DEVELOPMENT PROCESS
- Internal Development Standards and Procedures
- Software Engineering
- Software Development Tools and Facilities
- Software Test and Verification
- Software Documentation Approach
- Internal Independent Verification and Validation

PERSONNEL RESOURCES
- Estimating Software Personnel Requirements
- Manpower Needs and Qualifications
- Managing Software Personnel Resources
- Company Workload Profile

Ada TECHNOLOGY
- Management Process
- Development Process and Environment (Tool Set)
- Design Process and Methodology
- Personnel Skills and Qualifications
- Capability Demonstration and Risk Management

Table 8-2 SDCCR Factors

(a) Team Chief - Computer resources systems engineer or senior software engineer from the engineering staff;

(b) Program Manager/Project Manager;

(c) Software Manager;

(d) Software Engineer;

(e) Contracting Officer;

For smaller programs, it is possible for one individual to perform the program/project and software management role, and one individual to perform the chief/lead and software engineering role. In addition, it is desireable to include the following participation on the team:

(a) Product/Quality Assurance;

(b) Configuration/Data Management;

(c) Logistics;

(d) Cost Analyst;

(e) Contract Administrative Service/AFPRO/NAVPRO.

It is recognized that too large a team is counterproductive. The key is to perform the review with a small, knowledgeable group of software experienced acquisition personnel.

8.6 REFERENCES

1. ASD Pamphlet 800-5, "Software Development Capability and Capacity Review", HQ Aeronautical Systems Division, Wright-Patterson AFB, Ohio 45432, 10 September 1987.

CHAPTER 9
MANAGEMENT PRINCIPLES

9.1 INTRODUCTION

Managing software development is one of the biggest challenges facing today's government program manager (PM). The classic problems contributing to this challenge are illustrated in Figure 9-1. Some of these problems are beyond the control of the PM. For example, there is very little a PM can do about changing or growing requirements because of an evolving threat, or fuzzy requirements because of lack of details on the threat. Long lead times often force systems development to be initiated before the threat has been completely evaluated. Shooting at a moving taroe⁺ of changing or evolving requirements is a reality of all new systems acquisitions.

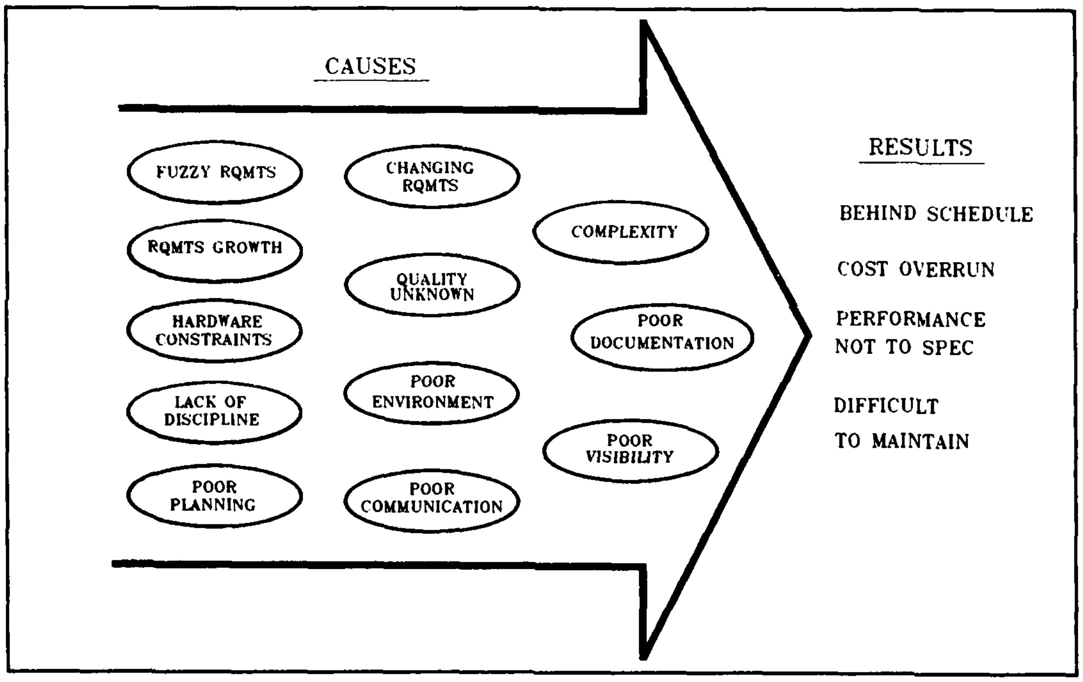


FIG. 9-1 Classic Software Development Problems

There are, however, many problems which can be minimized or eliminated by the PM. For example, there are certain actions that can be taken by a PM to eliminate poor engineering discipline, poor documentation, or poor planning. This chapter will address methods and principles which can help a PM deal with the complexity and difficulty of large scale software system development. In dealing with these problems, the common denominator is a disciplined systems engineering approach incorporating the principles of software engineering.

9.2 SOFTWARE ENGINEERING

The term software engineering is relatively new. It was first used in 1968 as the theme for several workshops associated with software development [1]. Today there is some debate as to what constitutes software engineering, but most people agree that applying the term software engineering to the software development process implies:

(a) The application of proven methods at each step of the development process which includes accepted practices, standards and procedures;

(b) The development and use of software tools and aids;

(c) The generation of specific documents during the various stages of the development process;

(d) The existence of a traceable path from the system requirements down to the final deliverable product.

For these proven methods to be effective in controlling software development, they must be accepted and practiced by the software developer. Mere proposal promises and contractual language is not sufficient. To achieve quality software which performs to specifications, is reliable, and is reasonably priced, these methods must be ingrained in the contractor's software organization. Table 9-1 lists some of the proven methods for software development.

All engineering disciplines develop tools and aids to help the practitioner apply the theory. Because software engineering is a relatively young discipline, it doesn't have the wealth of tools so readily available to its older sister disciplines. Much progress is being made, however, and currently available tools and aids include:

(a) Structured design and programming;

(b) Inspections and walkthroughs;

(c) Computer aided software engineering;

(d) Program design languages;

(e) The Ada programming language;

(f) Object Oriented Design and Programming.

Documenting each step of the software development process is absolutely essential. Not only does one need an effective way to communicate the rate of progress, one also needs a well documented final product. Since documentation is the only tangible evidence of the resulting product, documentation must be generated as one progresses through the various stages of software development. One cannot do a very effective job of generating software documentation after the fact.

* VISIBLE TRAINING AND EDUCATION PROGRAM
* INSTITUTIONALIZED PRACTICES AND PROCEDURES
* DEFINED ROLES WITH SYSTEMS, HARDWARE, AND TEST ORGANIZATIONS
* ADHERENCE TO A SOFTWARE DEVELOPMENT PLAN
* INSTITUTIONALIZED INTERNAL INSPECTION PROCEDURES
* STRUCTURED DESIGN AND PROGRAMMING
* PRODUCT ORIENTED WBS FOR FINANCIAL CONTROL
* INTERNAL INDEPENDENT VERIFICATION AND VALIDATION (IV&V)
* STRONG CONFIGURATION MANAGEMENT
    - SOFTWARE DEVELOPMENT LIBRARY
    - PROBLEM/TROUBLE REPORTING SYSTEM
* VISIBLE STATUS INFORMATION

Table 9-1 Proven Methods for Software Development

To ensure that the delivered product satisfies the requirements of the system, one must have a clearly definable path from the top level system requirements down to the various modules or units of code. This can be accomplished through the use of a requirements matrix which shows how each system level requirement is satisfied by a particular module or segment of code. This process can be automated.

9.3 GUIDELINES AND RULES

Over the past 30 years, certain guidelines and rules have evolved to help a PM successfully complete a software development program. One must keep in mind that the guidelines and rules given below are not just recipes or checklists to be blindly followed in order to ensure success. PMs must understand how to customize these guidelines and rules to the requirements of the program. They may choose to ignore any one of them as long as they are aware of the risks and have planned for dealing with them. The following guidelines provide the PM with software development planning and organizational principles which place software development in the perspective of the overall picture of system development:

(a) When planning and directing a program, PMs must make decisions based on a "system" perspective. They must review their alternatives and not allow either hardware or software to exclusively drive their decision. All decisions must consider the long term system effects.

(b) PMs must provide a forum for integrating the system development. Several techniques are available and should be part of both the government's and the contractor's review process. These techniques include stringent interface controls, reviews, and audits. All interfaces between software modules and between software and hardware must be clearly defined and strenuously controlled. Techniques such as reviews and audits will be discussed in the next section.

(c) Large investments up front can have significant leverage on reducing later system operation and support costs. Investing resources during the system and software requirements analysis phases can result in a better understanding of the user's requirements and a more stable baseline for design. Early expenditure of resources also provides the greatest amount of leverage in preventing errors. The bulk of the errors (30% to 70%) can be detected during the time when error correction is the cheapest [2].

(d) Software resource planning must remain stable once the program starts[1]. Software development has an inherent resource and schedule profile which means that the overall schedule cannot be stretched out without adversely affecting software development. More importantly, squeezing the schedule (trying to do a program too fast) can be a prescription for disaster. It is not always possible to add manpower to solve a software schedule slip [3].

(e) The software architecture should be the major driver in determining how the hardware is partitioned. In the past, PMs chose hardware first because they understood it best. Today the PM must consider the real-time requirements of the system and whether parallel processing and/or distributed processing will meet the overall requirements. In other words, the software architecture will often drive the hardware architecture.

(f) Identification of the software architecture must be performed simultaneously with the requirements definition and systems analysis tasks. In order to accomplish adequate hardware and software tradeoffs, the software must be viewed not only as the system integrator but as a system in its own right. Today's software development is usually in the critical path of the overall system development.

(g) The Program Manager should stimulate innovation and not be stifled by rules and regulation. "Think first,

---

[1] Portions of the remaining discussion in this section, Guidelines and Rules, are taken from a speech printed in Government Computer News and delivered by Dr. Kenneth Richardson, Executive Vice President of Hughes Aircraft Co. at a March 1988 National Security Industrial Association conference on software quality and reliability in Arlington, Va.

regulations second" should be the theme pursued by all system managers. New approaches should not be surreptitiously dismissed. For example, DOD-STD-2167A, "Defense System Software Development", is only a start and one must recognize that better approaches will evolve as more is learned about software development. This is not to say that DOD-STD-2167A stifles innovation. Although the standard embraces the "waterfall" approach to software development, it does allow for other advances such as rapid prototyping and evolutionary development.

(h) PMs must plan for growth and evolution. The software architecture and implementation should be specifically pointed toward maximum modularity, changeability, and growth potential. The PM's primary concern should be life cycle cost. As such, post deployment software support considerations need to be examined early during development to provide for cost effective lifetime support.

(i) The procurement process should allow for a flexible, robust, growable software design and find ways to reward innovative contractors. Some suggestions are to consider award fee or cost incentives based on operation and support after system delivery. The fees or incentives could be based on the ability to perform the intended functions, ease of support/maintenance, ease of modification, utility of documentation, and the effectiveness of the human interfaces. The intent should be to force the developer to focus on long term goals and a supportable systems. Also important, but more difficult to identify, are the front end decisions and methods that achieve these goals. In other words, it is what the developer does in the beginning that has the most influence on the end product. The developer must begin with a good process already in place.

9.4 PROCESS CONTROL

Process control is the key to achieving software quality. The process is the method used by the contractor for developing software. Achieving control of the process means that the process is predictable and measurable. A controlled process will minimize variability. How does a PM assess the contractor's process control system and procedures? Table 9-2 provides a series of questions developed by the Software Engineering Institute [4] as an aid in making this assessment. The PM must know that the process represents the contractor's commitment, philosophy, methodology, procedures, and standards for doing business. Process control and process management are principles of the Deming philosophy [5] for customer satisfaction. Application of Deming's philosophy requires the commitment of top management. That is why it is so important to select the right contractor; these values are not learned overnight.

* IS A MECHANISM USED FOR ENSURING TRACEABILITY BETWEEN THE SOFTWARE TOP-LEVEL AND DETAILED DESIGNS?
* ARE INTERNAL SOFTWARE DESIGN REVIEWS CONDUCTED?
* IS A MECHANISM USED FOR CONTROLLING CHANGES TO THE SOFTWARE DESIGN?
* IS A MECHANISM USED FOR ENSURING TRACEABILITY BETWEEN THE SOFTWARE DETAILED DESIGN AND THE CODE?
* ARE FORMAL RECORDS MAINTAINED OF UNIT (MODULE) DEVELOPMENT PROGRESS?
* ARE SOFTWARE CODE REVIEWS CONDUCTED?
* IS A MECHANISM USED FOR CONTROLLING CHANGES TO THE CODE? (WHO CAN MAKE CHANGES AND UNDER WHICH CIRCUMSTANCES?)
* IS A MECHANISM USED FOR CONFIGURATION MANAGEMENT OF THE SOFTWARE TOOLS USED IN THE DEVELOPMENT PROCESS?
* IS A MECHANISM USED FOR VERIFYING THAT THE SAMPLES EXAMINED BY SOFTWARE QUALITY ASSURANCE ARE TRULY REPRESENTATIVE OF THE WORK PERFORMED?
* IS THERE A MECHANISM FOR ASSURING THAT REGRESSION TESTING IS ROUTINELY PERFORMED?
* IS THERE A MECHANISM FOR ASSURING THE ADEQUACY OF REGRESSION TESTING?
* ARE FORMAL TEST CASE REVIEWS CONDUCTED?

Table 9-2 Process Control

What follows are some program management guidelines which focus on the day-to-day management of software development.

<u>All Software Tasks Must be Discrete</u> - This guideline is fundamental in determining how well the contractor can plan the effort. PMs must not allow level of effort or percent complete approaches because this closes the door on program progress visibility. The contractor must be able to define the work packages associated with the work breakdown structure (WBS) in sufficient detail to control and manage the effort. Each task should have a definite start, a definite end date, and a specific output. These tasks are normally on the order of 30 to 90 man-days in duration. Planning is usually accomplished as a rolling wave, the immediate six months or more are planned in detail with the remaining effort generally only visible at higher levels in the WBS. The <u>entire</u> effort must be totally scoped in time and resources at the very beginning of the project.

<u>Quantitative Requirements are Managed Through Margins</u> - Quantitative requirements lend themselves to measurable control methods. For example, computer memory and throughput are often tracked during development. In the beginning, estimates of software size and timing will be made and compared against target values to determine margins. Later, design language estimates can be made and these estimates will continue through the design process. As code is written,

actual measurements can be made. Early planning should allow for contractor and government margins with the use of a disciplined and documented control system. One approach is to baseline an estimate of the memory and throughput utilization on a monthly basis. An alert or trigger threshold value can initiate action should the threshold values be exceeded.

<u>Identify and Track Risk Areas</u> - The contractor and PM should be working as a team to manage program risk. An important ingredient of any program is to assess and reduce the risk as early as possible and before Milestone II. Risk management is an ongoing process. The first step is to identify risk areas, document them in the Software Development Plan, and devise a scheme for dealing with each risk item. These items are then tracked throughout development. A convenient method is to have a "Top Ten" list of risk areas that are tracked at least on a monthly basis along with a contingency plan for mitigating the identified risks. The plan should establish risk reduction objectives and schedules, assign responsibility and priority for risk reduction tasks, and develop a method for periodic reviews and assessments. Various management techniques to reduce risk have already been mentioned and include:

(a) Rapid prototyping;
(b) Incremental development;
(c) Internal (government) program reviews (at least monthly);
(d) Top Ten list review;
(e) Early demonstrations and testing of risk items;
(f) Government inspections and audits of the software development process.

Some potential problem areas include:

(a) Unrealistic cost and schedule;
(b) Vague or incomplete requirements;
(c) Inexperienced developers;
(d) Inadequate development environment (tools and methodology).

<u>Identify and Track Special Interest Items</u> - Special interest items such as Government Furnished Items (hardware, software and data) and subcontract items should also be tracked. Any items delivered to the contractor or received from the contractor are candidates for tracking. Certain critical internal deliveries such as code delivered for testing are candidates as well. The same approach used for tracking the risk items above can also be used to track special interest items.

<u>Requirements Must be Testable and Traceable</u> - Requirements must be testable in order to validate the system performance. In some cases, actual testing may be impractical

due to physical constraints, cost, or other considerations. When this is the case, system performance must be validated through inference or analysis and reflected in a testability matrix. Traceability is a key factor throughout development and becomes even more important during follow-on support. Requirements must be traceable from the system specification down through design, integration testing and DT&E. Traceability must occur in both directions--results of a test report must track back to the requirements. The only effective way to handle this for large programs is through some form of automation. Requirements traceability should be an integral part of the contractor's configuration management process.

Use Checklists for Design Reviews - Without adequate preparation design reviews can become nothing more than hectic "dog and pony shows". Design Reviews should be a major part of the software quality program. Program office personnel should be prepared for a review by arming themselves with appropriate analyses and a checklist of important and critical questions and adhering as much, as possible, to an agreed to formal agenda. The PM must do his homework through verification and analyses of critical areas that support the design approach. MIL-STD-1521B, "Technical Reviews and Audits for Systems, Equipments, and Computer Software", provides guidance on preparing checklists for various reviews. The checklists should also include special interest items and risk areas.

Formal Reviews Must be Viewed as Quality Gates - PMs must approach the formal review as a checkpoint for determining whether or not the project is ready to proceed to the next phase. A contractor should not be allowed to complete a design review if it hasn't satisfied all the requirements imposed by the design review. If the PM decides to proceed to the next phase with a less than satisfactory technical review, because of political or schedule considerations, the risks involved must be known and a contingency plan developed. Too often, however, the risks are much higher than perceived. It is essential to have a stable baseline since early mistakes become much harder and costlier (in time and money) to correct when they are discovered late in the development process. Proceeding before one is ready, usually increases the program risk.

Conduct Periodic Inspections and Audits - Periodic government inspections and audits can be useful when applied consistently. Inspections generally use a checklist to determine the specification and design completeness. Audits are similar to inspections with the additional factor of determining requirements traceability. These techniques can be valuable when applied to the interfaces. What better way to integrate the system than to ensure that the hardware and software properly communicate with each other? In addition to inspections, the contractor should also conduct walkthroughs as a standard business practice [6].

<u>Use Statistics Generated by Contractor Internal Reviews</u> - The contractor must have a system in place that includes a means of assessing the quality and progress of the work. The Software Development Plan should identify the software development system and indicate how government visibility will be provided. This is generally accomplished by contractually requiring the developer to provide the assessment data to the program office. The purpose is not for the government to manage the contractor's work (that's the contractor's job) but to communicate program development status and product quality. The contractor should gather statistics from the internal walkthroughs and inspections and use this data to manage the software development. The mechanisms the contractor has in place for software development constitutes the process control system. The contractor's software process control system is analogous to a manufacturing process control. Statistics at this level will provide valuable trends and indicators to program management.

<u>Integration Must be Visible on Master Schedule</u> - Integration brings the interfaces together. They occur at all levels: software to software, software to hardware and software to systems. Integration and integration testing must be planned and be highly visible to the government PM. It is important that the development of hardware and software be approached as a maturing process strategy. Pieces of the system should be brought together in a planned, logical fashion. Some level of confidence in both the hardware and software components must be established before proceeding with higher level integration. If this isn't done, problems that occur during integration will be difficult to diagnose since they may be in either component--hardware or software. Integration of the system, particularly the critical components, are likely candidates for special interest or risk item tracking.

9.5 REQUIREMENTS AND PROTOTYPING

The PM can have a major impact on the success of the program during the requirements definition phase. Investment of time and resources in requirements definition has the biggest program payoff in both schedule and cost. Clear and unambiguous requirements, however, seem to be the most elusive ingredient in any program. Since software development is an exercise in abstraction and an intellectual process that must be captured in some tangible form, how does one approach requirements definitions? Two tools available to aid the process are formal specification development tools and rapid prototyping.

9.5.1 Specification Development Tools

Since specifications are generally written in prose, they suffer from significant interpretation and traceability problems as well as being very error prone. Formal specification development tools such as those listed below are designed to alleviate these problems. Because the formal specification language is much more specific than prose, it can be automated. This provides the use of computer assistance to check for errors, completeness, consistency, and traceability. Some commonly used formal specification development tools are:

PSL/PSA	Problem Statement Language/Problem Statement Analyzer: Originally developed for data processing applications. Widely used in other applications [7].
RSL/REVS	Requirements Statement Language/Requirements Engineering Validation System: Real-time process control [8].
SADT	Structured Analysis and Design Technique: Interconnection structure for any large, complex system. Not restricted to software systems [9].
SSA	Structured System Analysis: Gane and Sarson version used in data processing applications that have database requirements. DeMarco version suited to data flow analysis of software systems [10,11].
Gist	Textural language developed at USC/Information Sciences Institute: Object-oriented specification and design. Refinement of specifications into source code [12].

9.5.2 Rapid Prototyping

Rapid prototyping may be the most powerful tool available to analyze and refine requirements and should be encouraged as a requirements definition tool from the very beginning. The PM should incorporate rapid prototyping as part of the contract.

A software rapid prototype is an analytical tool for refining software requirements. It is used during the requirements analysis phase with minimal constraints on choice of programming languages, documentation, and use of standards. In essence, it entails the almost unconstrained development of a software package with the primary goal of achieving quick results. The objective is to compare these quick results with the initial system requirements. By allowing the user a quick look at the potential end product, they will be better able to answer the questions "Is this what you wanted?" or "Is this what you meant?" Documentation is minimal and it is not deliverable. Once a decision is made to deliver, it ceases to be a rapid prototype. This is analogous to the hardware requirements and design process of using engineering models, breadboards and brassboards. This is not to say that one would not use an approach of developing software using a "prototype" (as opposed to a "rapid prototype") for delivery and operation. This is yet another method called evolutionary development and it will be discussed later in this chapter. The value of a rapid prototype is in the capability to better communicate software requirements during the requirements analysis process. What better way to communicate with the users than to present them with an artifact that represents the system? Often the rapid prototype will represent the user interface (controls and displays) providing for both input and output. It may execute representative scenarios or operational profiles to determine the validity of the specifications. The prototype can be used throughout development, including Full Scale Development, to test out design concepts as well as develop a priori test results for future testing of the actual product. One must remember that a rapid prototype does not evolve into a deliverable software product. It should be discarded once it is used to clarify and refine the requirements.

9.5.3 Incremental and Evolutionary Development

Incremental and evolutionary development are techniques for dealing with large, complex systems. Figure 9-2 illustrates this approach. Design begins after the system and software requirements have been baselined.

The objective of incremental development is to produce a complex software product by building the total system capability in ever increasing increments. The second software delivery or increment will have more capability than the first delivery, the third more than the second and so on. As the saying goes "If you have to eat an elephant, eat him one bite at a time". Software development for Inter-Continental Ballistic Missiles (ICBMs), for example, is developed this way. The Operational Flight Program (OFP) may be designed and developed in three increments. This incremental approach must be preplanned with the overall development strategy and test plan. The first system level capability to be demonstrated is the capability to fly from point "A" to point "B". This requires the appropriate planning for developing and integrating both hardware and software components in support of the scheduled events. The first increment would have the basic flight control, navigation, guidance, and task control functions. The next increment would add the system capability for multiple reentry vehicle deployment. Again, meticulous planning would have preceded this activity to make sure that the appropriate hardware and software components are designed and developed on an integrated schedule. For the software,

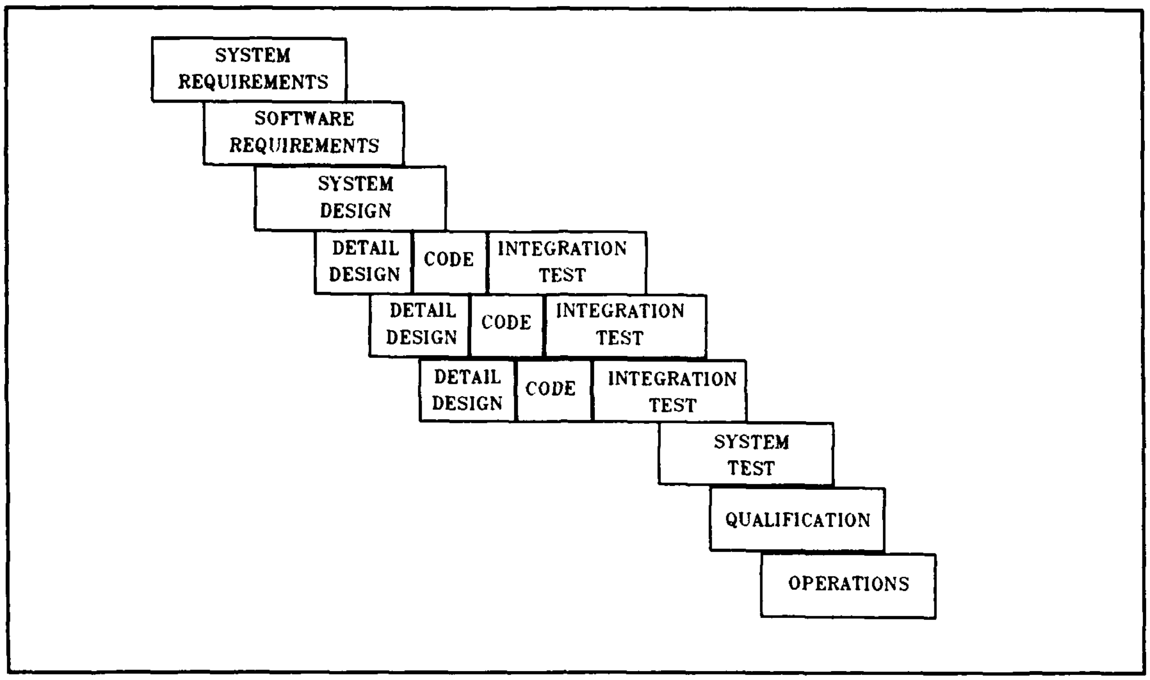


Fig. 9-2 Incremental Development

this second increment or "build" would have the same capabilities as the first increment plus the capability to deploy multiple reentry vehicles during a flight. The final software increment may contain maintenance diagnostic capability and continuous navigation instrument calibration capability. One begins with a minimal program and then adds additional capability until the complete system is developed.

Evolutionary development is very similar to incremental development but it is more long term. While incremental development occurs during a single development phase, typically during FSD, evolutionary development can occur over several phases or be part of a preplanned product improvement approach. Evolutionary development is the recommended approach for large Command, Control, Communications and Intelligence (C^3I) systems. Rapid prototyping can be combined with this approach to help define the requirements. The evolutionary approach helps to deal with "fuzzy" requirements where the general requirements are known but the details are lacking. For example, in a large C^3I program the PM may know that he must design a system to communicate with multiple users, be able to react to rapidly changing threats, and be able to adapt to the environment in real-time. He may not, however, know the requirements of all the users, who may in turn not know themselves until they are able to work with the actual equipment in a scenario or simulated environment. The evolutionary development approach would develop an early model with flexibility and growth specifically as part of its design. This model would then be used under actual or simulated conditions with feedback from actual operation cycled back to update the requirements. This approach could continue indefinitely.

9.6 SUMMARY

The problems associated with managing software development can be overwhelming. But software development is manageable. This chapter has discussed some of the tools available to the program manager to help in planning a development strategy as well as to assist in the daily management of the program. The Program Management Checklist in Figure 9-3 serves as an additional reminder of the key elements that have been discussed throughout this text.

* Plan
  -- Cost and schedule
  -- Development and support
* Select tangible inchstones
* Review schedule after requirements are defined
* Scrub requirements
* Build rapid prototypes
* Create/Update S/W size and cost estimates
* *Design within systemm" constraints*
* Don't let hardware needlessly constrain software development
* Establish CRWG and provide for software support
* Beware of government furnished products and subcontracts
* Strongly consider incremental development for large developments
* Create and manage margin (schedule/memory/throughput)
* Understand the contractor's development process
* Establish an internal control system
* Use software metrics
* Select resources and language
* If performing IV&V, then do it early
* Build SIL/SIF

Fig. 9-3 Program Management Checklist

9.7 REFERENCES

1. Fairley, Richard E., "Software Engineering Concepts", Tyngsboro, Mass., McGraw Hill Book Co., 1985.

2. McCabe, Thomas J. and G. Gordon Schulmeyer, "The Pareto Principle Applied to Software Quality Assurance", Handbook of Software Quality Assurance, Ed. G. Gordon Schulmeyer and James I. McManus, New York, NY, Van Norstrand Reinhold Company Inc., 1987.

3. Brooks, Frederick P., "The Mythical Man-Month: Essays on Software Engineering", Addison Wesley, July 1978, Second Printing.

4. "A Method for Assessing the Software Engineering Capability of Contractors", Software Engineering Institute, September 1987.

5. Deming, W. Edwards,"Out of the Crisis",

6. Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development", IBM Journal, Vol. 15, No.3, 1976.

7. Teichow, D. and Hershey, E., "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems", Transactions Software Engineering, Vol. SE-3, No. 1, January 1977.

8. Alford, M., "A Requirements Engineering Methodology of Real-Time Processing Requirements", Transactions Software Engineering, Vol. SE-3, No. 1, January 1977.

9. Ross, D., "Structured Analysis (SA): A Language for Communicating Ideas", Transactions Software Engineering, Vol. SE-3, No. 1, January 1977.

10. Gane, C. and T. Sarson, "Structured Systems Analysis: Tools and Techniques", Prentice-Hall, Engle Cliffs, NJ, 1972.

11. DeMarco, T., "Structured Analysis and System Specification", Yourdon Press, New York, 1978.

12. Balzer, R., "Gist Final Report", Information Sciences Institute, University of Southern California, February 1981.

CHAPTER 10
SOFTWARE CONFIGURATION MANAGEMENT

10.1 INTRODUCTION

The dictionary defines configuration as the "relative disposition of the parts or elements of an item" and defines management as the "act or manner of handling, directing, or controlling" [1]. Configuration management, then, can be generally defined as the act of controlling all elements of a particular item. When applied to weapon systems, configuration management is the systems engineering management process that identifies the functional and physical characteristics of system components, controls changes to those characteristics, and records the status of the changes implemented. It is the means through which the integrity and continuity of the design, engineering, and cost trade-off decisions made between technical performance, producibility, operability, and supportability are reported, communicated, and controlled [2].

Software configuration management is formally defined as the discipline of identifying the software configuration of a system at discrete points in time for the purposes of systematically controlling changes to this configuration and maintaining the integrity and traceability of this configuration throughout the system life cycle. This chapter explains the basic elements that constitute the discipline of software configuration management and shows how the application of these elements to the software development cycle facilitates the transformation of software into a visible, manageable entity [3]. Figure 10-1 illustrates the interrelationship of the four functions of software configuration management (SCM): identification, control, status accounting, and audits.

Both the developer (a contractor or another government agency) and the government procuring agency may apply configuration management procedures to a specific development program. This chapter will concentrate on the government's configuration management practices as described in DOD-STD-480A, "Configuration Control, Engineering Changes, Deviations, and Waivers", and MIl-STD-483A, "Configuration Management Practices for Systems, Equipment, Munitions, and Computer Resources". Whenever it becomes necessary to illustrate a specific point, the contractor's configuration management practices will be referenced.

10.2 CONFIGURATION IDENTIFICATION

Configuration identification determines how the software system will be divided for ease in managing and controlling change. The configuration of a system is comprised of a physical and a functional configuration. Physical configuration refers to the detailed design or physical attributes of a system or item. It is normally described by hardware drawings and software code listings. Functional configuration addresses the functions a system or unit performs. It is primarily established by hardware and software requirements documents [3].

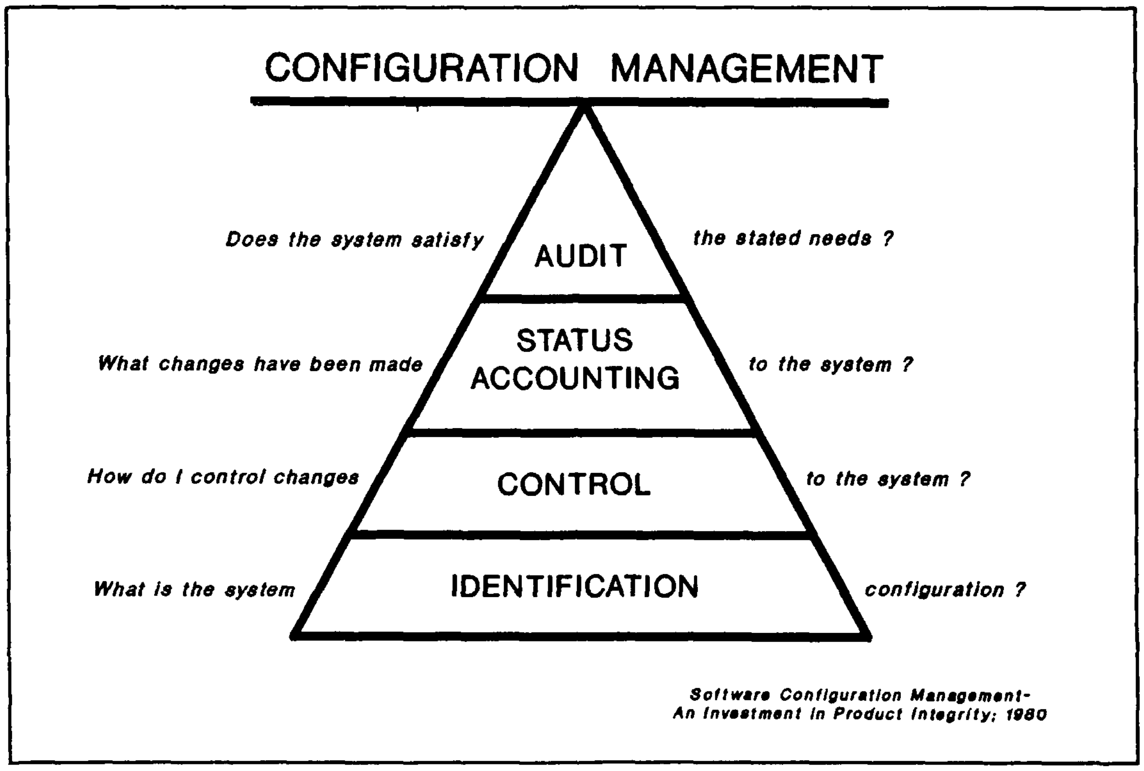


Fig. 10-1 Configuration Management Functions

For software, configuration identification actually specifies and identifies all components of a software system throughout its life cycle, from the development of specifications to the generation of actual code [4]. Computer software configuration items (CSCIs), along with their functional and physical characteristics, will normally be determined during the Demonstration and Validation (D/V) phase of the acquisition life cycle and prior to the System Design Review (SDR). MIL-STD-483A, Appendix XVII, provides the following guidance for selecting configuration items (CIs):

(a) Select configuration items based on functional and performance parameters which must be controlled to satisfy an overall end use function (e.g., defensive avionics system software);

(b) Select configuration items which require an optimum level of government control during acquisition (e.g., code verification software);

(c) Select configuration items based on the need to control a CI's inherent characteristics or to control that

CI's interface with other CIs (e.g., controls and displays software);

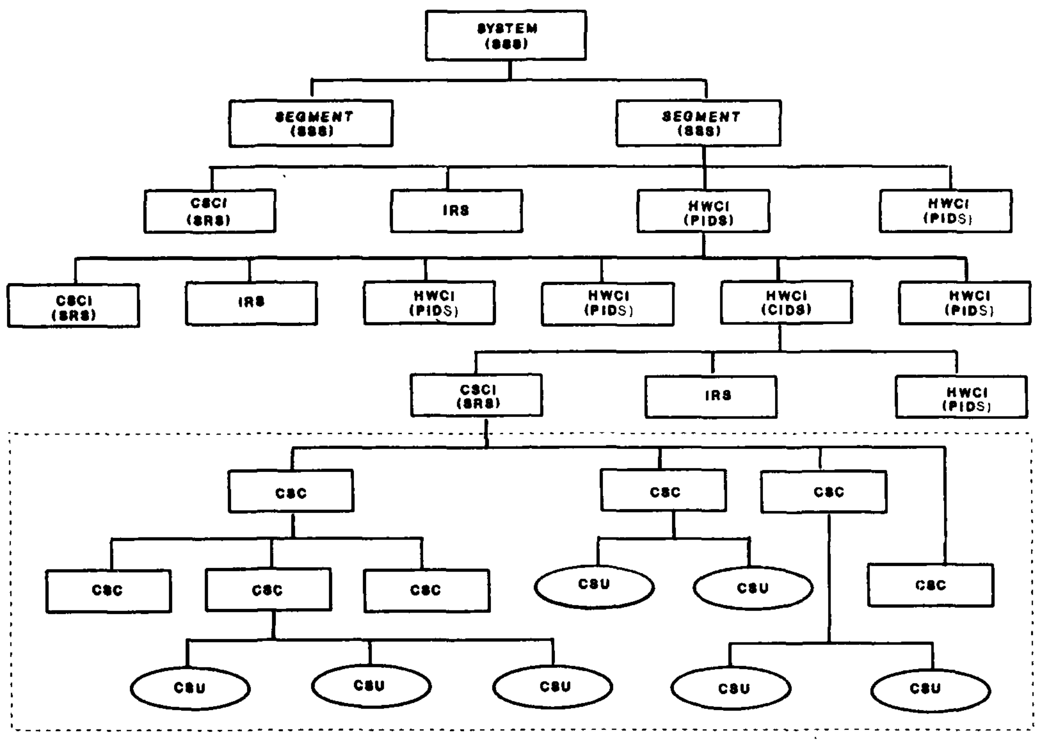
(d) When selecting CIs, evaluate other factors such as schedule, the engineering release system, financial impact, and new, modified, or existing design parameters.

One has to be careful in selecting the number of configuration items. Too many configuration items may increase the management and administrative efforts required to adequately track and control the status of the CSCIs. This additional effort may delay the schedule and increase the cost of the software development. On the other hand, too few CSCIs may not only minimize the program office's visibility into the software development process, but tend to reduce control of the software design and possibly lead to operational deficiencies.

Configuration identification provides a means for isolating the system components as a basis for controlling their development. There are three steps to software configuration identification: first, the software system must be broken down into a number of known manageable parts or CSCIs; second, these parts must be uniquely named; and third, as these parts change with time, the various versions that appear must also be uniquely identified. The first step is closely associated with the processes of specification, analysis, and design. The other two steps require rigorously enforced standards and procedures [4]. Figure 10-2 represents a generic breakdown of a software system into its various parts of CSCIs, computer software components (CSCs), and computer software units (CSUs). The contractor will normally propose a list of CSCIs based on the design requirements, management structure, and available resources. These selections will be included in the Software Development Plan and reflected in the Configuration Management Plan along with the contractor's configuration management methodology. These documents are delivered to the government for approval in accordance with the provisions of the Contract Data Requirements List (CDRL).

10.3 CONFIGURATION CONTROL

Once the system configuration is established, the next logical step is to devise a method to control changes to that configuration. Unlike hardware, software is an intangible product difficult to "see" and more difficult to manage. Software can still be properly managed, however, by imposing configuration control methods on its development process and on the resultant products of that process. Edward Bersoff defines software configuration control as "...the orchestration of the processes by which the software portion of a system can achieve and maintain visibility throughout its journey through the life cycle. It provides the tools (i.e., documentation, procedures, and an organizational body) to control the system implementation as well as changes to it" [4].



LEGEND

CIDS	Critical Item Development Spec	PIDS	Prime Item Development Spec
CSC	Computer Software Component	SSS	System/Segment Spec
CSU	Computer Software Unit	SRS	Software Requirements Spec
IRS	Interface Requirements Spec		

Fig. 10-2 Breakdown of Software System

The configuration control process includes the following areas: interface control, baseline management, configuration control boards (CCBs), software configuration review boards (SCRB), software problem reports (SPRs), and software engineering change proposals (ECPs). As previously stated, once a "baseline" has been established or identified, the next step is to monitor any change to that baseline, maintain a record of that change, and update the necessary elements of that change.

Before discussing the configuration control process, one has to understand the differences between the two categories of change proposals: Class I and Class II changes. Class I changes affect form, fit, or function; although other factors, such as cost or schedule, can cause a Class I change. All Class I changes must be submitted to the CCB for approval. All other changes are Class II changes. Examples of Class II changes are editorial changes in documentation or minor changes which don't affect the established baselines. Class II changes only require government concurrence on the classification. This is usually accomplished by government plant representatives. A more detailed discussion relating to software Class I and II changes can be found in Appendix XIV to MIL-STD-483A.

10.3.1 Interface Control

An interface is defined as the functional or physical characteristics which serve as a common boundary between two or more items. With respect to system development, these boundaries are found between hardware and software, hardware and hardware, and software and software configuration items. The interfaces are defined by mechanical and electrical characteristics, reliability and maintainability requirements and software format, timing, and programming languages requirements. Interoperability requirements with other systems and subsystems will also define the system under development.

Due to the increasing complexity of today's systems, the program manager must establish a government Interface Control Working Group (ICWG) during the D/V phase. The ICWG is usually chaired by the procuring agency's engineering representative and is comprised of representatives from the development organizations, user organizations, and software support activities as shown in Figure 10-3. The ICWG, in

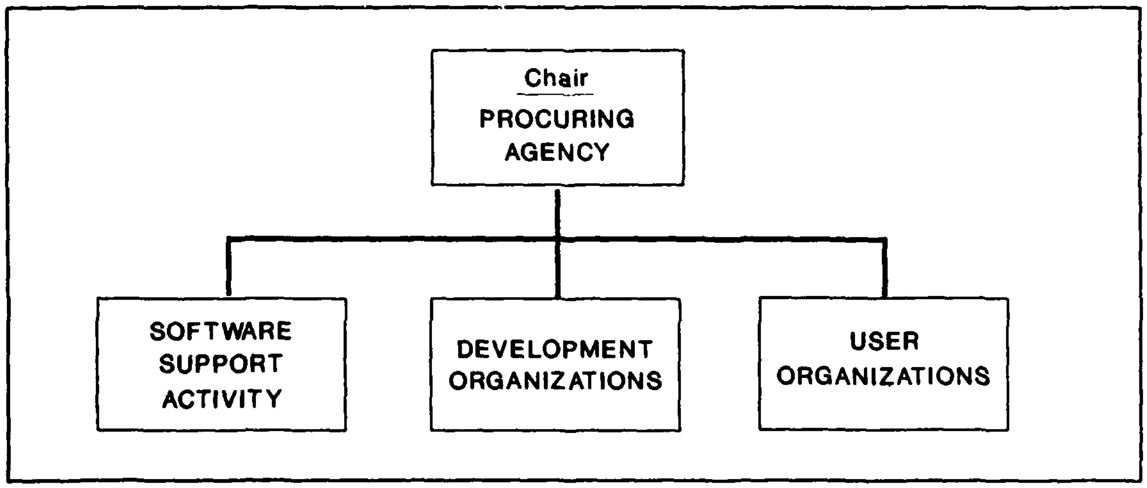


Fig. 10-3 Interface Control Working Group

coordination with the Computer Resources Working Group (CRWG), defines current and proposed software and hardware interfaces, obtains and assesses quantitative data on the interfaces, and investigates interoperability requirements between various systems and subsystems.

The prime contractor will also establish an ICWG to address the system interface requirements, especially when more than one contractor is involved in developing the weapon system. The contractor's ICWG will usually be chaired by an individual from the engineering organization, with all associate contractors as members and the responsible DOD agency as an observer. The ICWG documents agreements on the interfaces affecting CSCIs and hardware configuration items (HWCIs) involving more than one contractor. The contractor's ICWG should review all interface changes before submitting then to the Program Office [5].

10.3.2 Baseline Management

Several phases have been established to monitor the progress of a weapon systems acquisition. Once specific objectives have been achieved for each phase, the acquisition process can proceed into the next phase of the life cycle. Specific reviews have been established to examine the appropriate documentation such as specifications, design documents, and test reports for correctness and requirements traceability. Once these documents are approved, a baseline is established to allow the performance of both the developer and the system to be measured. As the baseline is changed, the provisions to the contract require that the system integrity be maintained throughout the development process. The bottom line of baseline management is the establishment of a common technical relationship between the developing agency and the developer. In the case of software configuration management, that relationship is based on the CSCI.

There are three baselines associated with the acquisition of a weapon system: functional, allocated, and product baselines. The functional baseline is typically established at the completion of the System Design Review (SDR) upon approval of the Type A system specification. Formal configuration control for the system is initiated once the system specification has been approved. The allocated baseline for the system is established once the allocated baseline for all configuration items is determined and the Type B specifications for each configuration item have been approved. For hardware, this normally occurs at the Preliminary Design Review (PDR), but no later than the Critical Design Review (CDR). For software, the allocated baseline is established at the completion of the Software Specification Review (SSR). The product baseline is typically established at the completion of the Physical Configuration Audit (PCA) after the Type C specifications are approved.

10.3.3 Configuration Control Board

The Configuration Control Board is the organizational body within the development organization responsible for formal processing of changes to established baselines. The

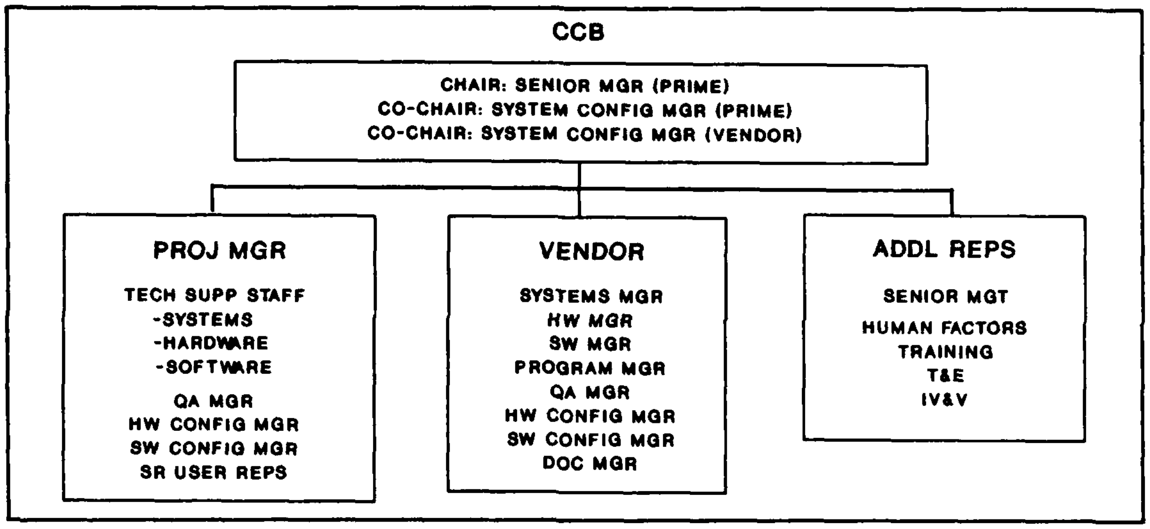


Fig. 10-4 Contractor CCB

function of the CCB is to approve, monitor, and control changes to the system. Figures 10-4 and 10-5 illustrate typical membership of government and contractor CCBs. The composition of both CCBs are very similar; however, the responsibility of the government CCB is at the system level. The contractor has additional configuration control responsibilities at lower hierarchical levels for both software and hardware as depicted in Figure 10-6 [3].

10.3.4 Software Configuration Review Board

The Software Configuration Review Board (SCRB) reviews and evaluates all proposed changes to the software baselines and determines the processing and disposition of software problem reports. The SCRB serves as a filter for the CCB on software related matters. Software problem reports, incident reports, change requests, and change proposals will first be submitted to the SCRB for review and evaluation. The SCRB will determine if the pending software changes should be disapproved or forwarded to the CCB for formal approval as baseline changes. Prior to its decision, the SCRB would have reviewed a number of proposed software changes, prioritized their urgency, and determined if the changes should be made singly or in compatible groupings or block changes.

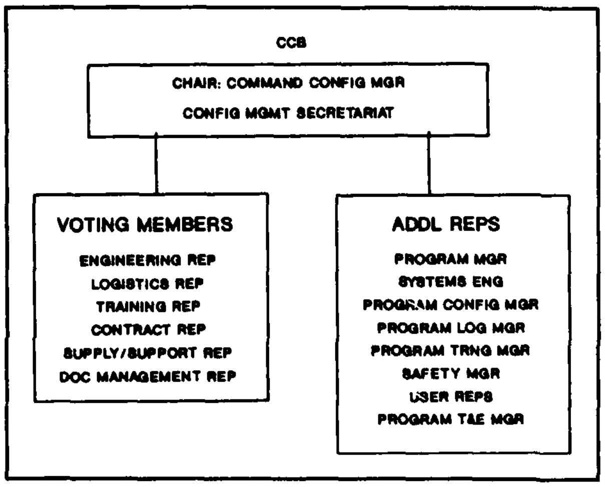


Fig. 10-5 Government CCB

As with the Configuration Control Board, the SCRB is also used by both the government and the contractor. Figure 10-7 illustrates the typical composition of these two SCRBs.

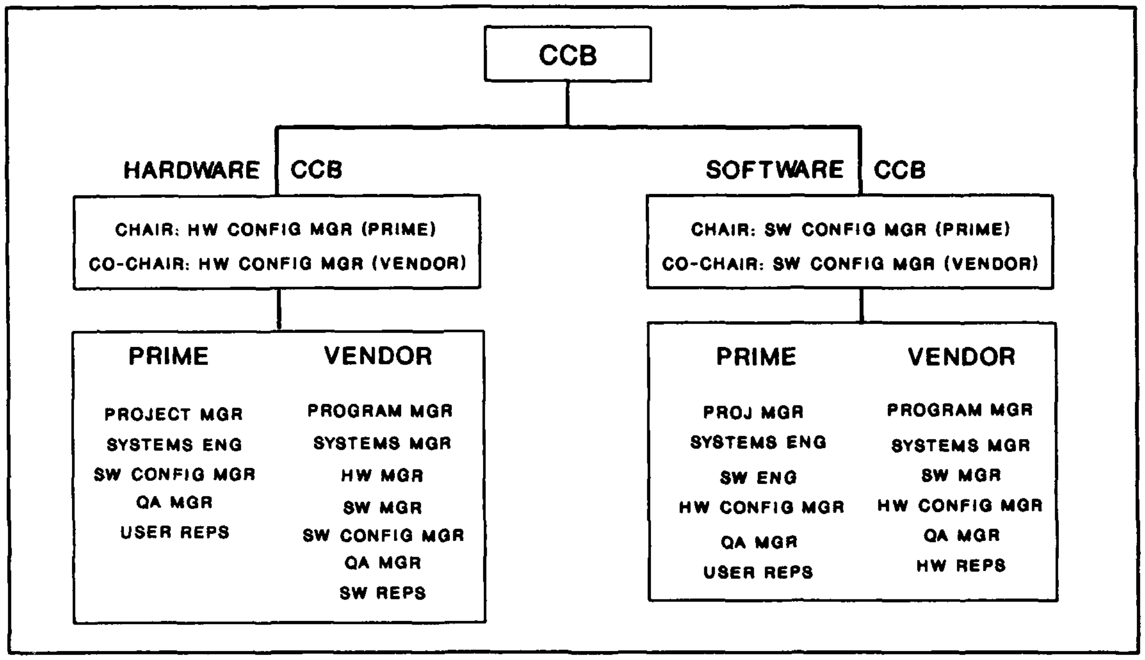


Fig. 10-6 Configuration Control Board

In the government, the SCRB normally functions during the production/support phases of the acquisition cycle of a weapon system. That is one of the reasons for including the Software Support Activity as part of its membership. During the transition to organic support, the program manager may choose to have the contractor as a non-voting member of the SCRB. As a minimum, the SCRB should be conducted annually in conjunction with planned upgrades (or releases) of the software system. However, the SCRB may be convened at any time for major (critical) deficiencies discovered in the software. The Software Support Activity or contractor may be assigned the action to resolve these deficiencies and change proposals will be submitted to the CCB for approval.

The software developer normally employs the SCRB during development. Since the SCRB is a software board, its membership is made up of mostly software oriented representatives. The software manager chairs the board and has final approval authority. The SCRB should meet routinely to act on all submitted software problem reports (SPRs). Any change that effects program baselines must be transmitted to the CCB for final disposition [6].

A key member of the developer's SCRB is the software development library (SDL) librarian who acts as the recorder for the board. Since the SCRB is administered and supported through the SDL, the SDL librarian is responsible for tracking the status of the SPRs, forwarding the recommended SPRs to the CCB for approval, incorporating the approved changes in the

SDL, and maintaining up-to-date records of these reports.

10.3.5 Configuration Control Process

The configuration control process (Figure 10-8) is a very time consuming and active process. It is the heart of the configuration control function. This section will concentrate on the processing and controlling of software changes. Both the government and the developer follow a similar internal change control process which results in software ECPs forwarded to the government CCB for evaluation and approval.

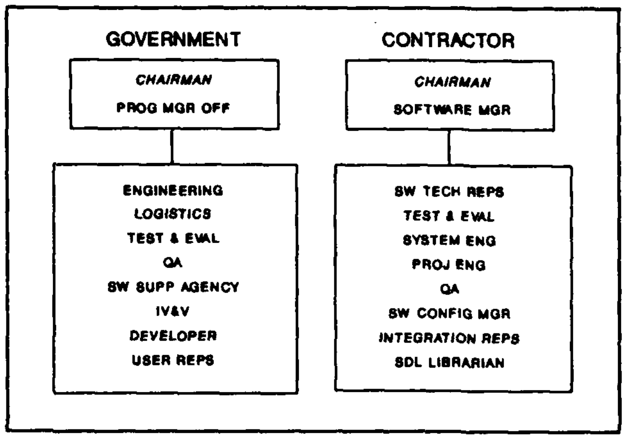


Fig. 10-7 Software Configuration Review Board

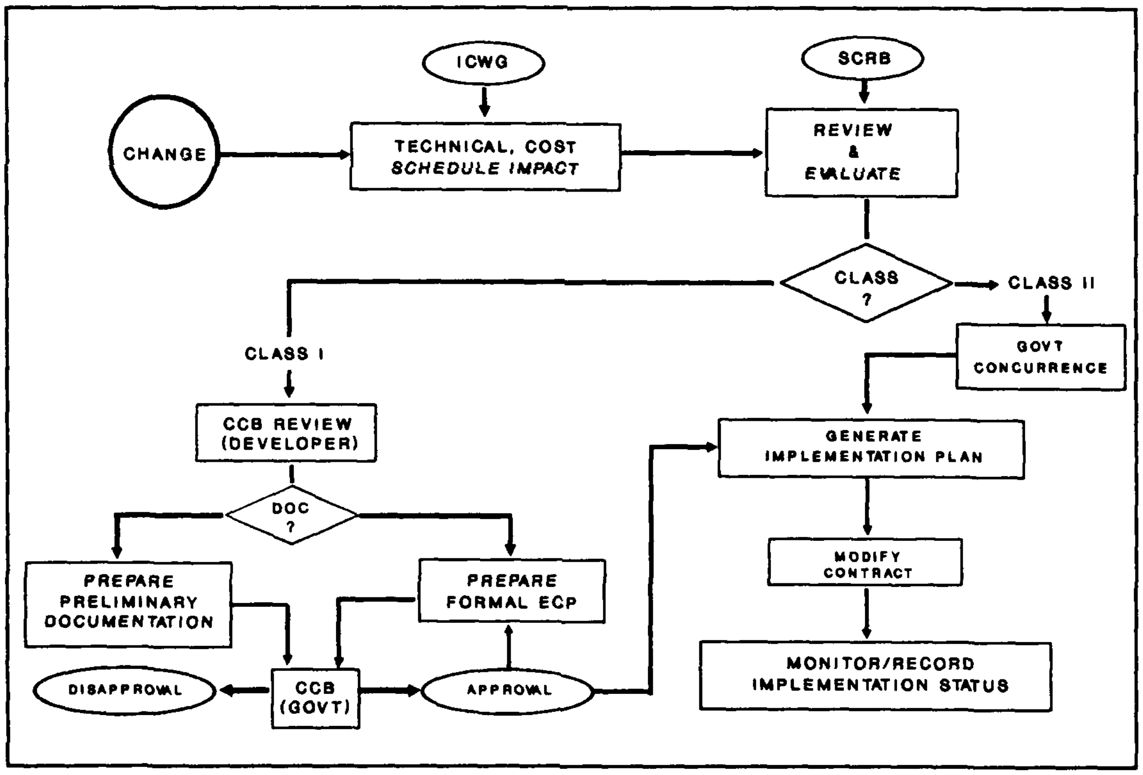


Fig. 10-8 Change Control Process

The configuration control process begins with the initialization of a change to an established baseline (e.g., SRS, IRS). These changes may be initiated by government direction, ICWG activity, or contractor/subcontractor activity. The cognizant engineering activity (developer), in coordination with its ICWG, will perform impact analyses on the technical, cost, and schedule aspects of the proposed change. Once all changes, including software changes, have been reviewed and evaluated, the developer will categorize them as either Class I or II changes. Any software related issues will be forwarded to the SCRB for review, evaluation, and disposition.

Class II changes will be submitted to the government for concurrence on the classification. The government's plant representative (AFPRO, NAVPRO, or DCASPRO) may approve the classification and sign off on the change. However, it is strongly recommended that the program manager work with the prime contractor to establish an informal body of government technical representatives to be located at/near the plant representative office. Their function would be to assist the plant representative offices in evaluating proposed changes and to provide liaison between the engineering disciplines of both the government and developer. Once the classification of a Class II change is approved by the government, the contractor may implement and monitor the status of the change. Class II changes which do not receive government concurrence may be submitted as Class I changes.

Class I changes follow a more formal route through the CCB. The developer will submit all proposed Class I changes to its CCB for review and disposition. The CCB will determine if the change should be formally submitted to the government. If they decide not to, they may choose to follow an informal route by submitting preliminary documentation consisting of an Advanced Change/Study Notice, an Engineering Change Request, or a preliminary ECP. If this informal documentation is approved, the government will notify the developer to submit formal documentation. The formal documentation may consist of an ECP, a Specification Change Notice with specification page changes, a Request for Deviation/Waiver, an Interface Revision Notice, and supporting cost data. Once approved, the program office will notify the contractor and both will monitor the implementation status.

10.4 CONFIGURATION STATUS ACCOUNTING

Configuration status accounting is the management information system that provides traceability of changes to configuration baselines and facilitates the effective implementation of changes. It consists of reports and records documenting change actions affecting CSCIs. The basic documentation includes the Configuration Identification Index and Status Accounting Report which describe the current and approved configuration. MIL-STD-482A standardizes data elements with regard to format, frequency, and record keeping [2]. For software, the Software Development Library is one of the key developmental configuration management tools used by the contractor for status accounting of software changes on all software products.

10.4.1 Software Development Library

The contractor's Software Development Library (SDL) is defined in DOD-STD-2167A as a controlled collection of software, documentation, and associated tools and procedures used to facilitate the orderly development and subsequent support of software. An SDL provides storage of, and controlled access to, software and documentation in human-readable form, machine-readable form, or both. The SDL maintains established project baselines, and monitors and controls the project development configuration baselines and data products. Software products consist of documentation and listings, source code, executable (machine) code, and status records for all major software elements. The SDL responsibilities consist of:

(a) Technical control and project monitoring of baseline content and quality;

(b) Technical control and project monitoring of the development configuration;

(c) Organizational facilities for baselining and controlling the content of structure or software products;

(d) Reporting procedures for software design or implementation issues and documentation of library contents of data products [5].

The SDL librarian is responsible for managing the software development library. The librarian stores the original software products in the SDL, maintains current listings and copies of these products, updates them as required, and distributes copies to appropriate personnel. As recorder for the SCRB, the SDL librarian schedules and coordinates the SCRB meetings and reviews the software problem reports (SPRs) to insure that these reports are ready for discussion at the SCRB. The librarian tracks, monitors, and records the status of action items assigned to the SPRs, and prepares and distributes the minutes of SCRB meetings.

10.4.2 Software Development Folder

A Software Development Folder (SDF) is defined in DOD-STD-2167A as a repository for a collection of material pertinent to the development configuration. The contents of the SDF typically include (either directly or by reference) design considerations and constraints, design documentation and data, schedules and status information, test requirements, test cases, test procedures, and test results. The contractor shall document the development of each computer software unit (CSU), computer software component (CSC), and CSCI in SDFs. He will also establish a separate SDF for each CSU, CSC, and CSCI or logically related groups of CSUs or CSCs. The SDFs shall be maintained for the duration of the contract. The SDFs will be part of the SDL and be made available for government review upon request.

10.5 CONFIGURATION AUDITS

The fourth function of configuration management is to perform a set of configuration audits to verify that the selected configuration items conform to the specifications and related technical data. There are two types of audits conducted: a Functional Configuration Audit (FCA) and a Physical Configuration Audit (PCA). The details of these audits are described in MIL-STD-1521B, "Technical Reviews and Audits for Systems, Equipments, and Computer Software".

The software FCA is a formal examination of the functional characteristics of a CSCI prior to acceptance to verify that the CSCI has achieved the performance specified in its Software Requirements Specification (SRS) and Interface Requirements Specification (IRS). Other technical documentation such as the Software Test Plan (STP), the Software Test Descriptions (STD), the Software Test Reports (STR), and minutes of the design reviews are evaluated for completeness. The FCA is a means of validating that the development of a CSCI has been satisfactorily completed and that the CSCI performs as required.

The PCA is a formal examination of the "as-built" version of the CSCI as described in the Software Product Specification (SPS). The source code for each CSCI is compared with the associated documentation (SRS, IRS, software design documents (SDD), Interface Design Documents (IDD), and Version Description Documents (VDD)) for accuracy and completeness. The PCA is a means of establishing the product baseline for each CSCI and generally occurs prior to system integration and testing. However, system level PCAs are often delayed until after system integration and testing is completed. The PCA also verifies that the tested object code can be recreated or compiled from the baselined source code.

10.6 SUMMARY

Configuration management is essential to the development of mission critical computer resources; it provides the needed program visibility. It allows for the control of the software requirements, design, and final product. It defines the limits imposed on and by the contractor during the development of a software product. Configuration management is especially important for software since the physical and functional characteristics of software cannot be assessed by visual inspection like hardware [6]. Embracing the configuration management discipline means making a continuous, firm commitment to the principle of keeping track of the current status of information that is changing [7].

Without the structure of a sound software configuration management program, the development of complex software systems would be impossible. Sound configuration management is extremely important for life cycle support of software.

10.7 REFERENCES

1. College Dictionary, Random House Inc., New York, NY, 1968.

2. "Configuration Management", System Engineering Management Guide, Defense Systems Management College, Ft. Belvoir, VA, October 1986.

3. DOD Directive 5010.19, "Configuration Management", 1 May 1979.

4. Bersoff, Edward H. Software Configuration Management, An Investment in Product Integrity, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980.

5. Buckle, J. K., Software Configuration Management, McMillan Press, Ltd., 1982.

6. Ferens, Daniel V. Mission Critical Computer Software Support Management, Air Force Institute of Technology, School of Systems and Logistics, Wright-Patterson AFB, OH., First Edition, May 1987.

7. Evans, Michael W., Software Quality Assurance and Management, John Wiley and Sons, 1987.

CHAPTER 11
INDEPENDENT VERIFICATION AND VALIDATION

11.1 BACKGROUND

Independent Verification and Validation (IV&V) of software was born during the early days of the space and missile programs. Both NASA and the military realized that software developed for spacecraft and missiles had to perform correctly the first time. For example, if the software did not perform as required during the launch phase of a missile, there were no second chances. Software failures usually meant the loss of the mission, the launch vehicle, and the personnel on board.

Because of the importance and criticality of spacecraft software, money and time were set aside for an independent organization or contractor, other than the software developer, to perform several functions:

(a) Independently test the performance of the developed software;

(b) Ascertain that the developed software satisfied all of the system level requirements;

(c) Independently develop a separate software package for those functions deemed to be critical to the success of the mission.

This type of IV&V effort did not come cheap! A full blown IV&V effort could sometimes exceed 50 percent of the software development cost [1]. The criticality of the missions, however, justified these kinds of expenditures. The same was not true when the IV&V philosophy was introduced into non-missile weapon system programs. Many programs failed to tailor the IV&V approach of the space and missile programs to the requirements of their own programs. Much of the software developed for aircraft, tanks, and ships could be classified as noncritical. This meant that many software failures had no impact on critical subsystems and did not affect crew safety. For example, a failure in a built-in-test (BIT) or other diagnostic on-line subsystem, usually did not have any bearing on the success or safety of a particular mission. Likewise, failure in a piece of automatic test equipment (ATE) or in a data reduction program was not catastrophic. Unfortunately, large amounts of program funds were needlessly spent in the IV&V of non-critical or secondary software. This has resulted in two schools of thought. One school feels that software IV&V is an unnecessary expense for most weapon systems while the other school feels that IV&V should be performed on all software developments. Neither is entirely true! Software IV&V is indeed an important aspect of developing quality software; but it has to be focused on those areas that are critical to the success or safety of a mission.

Although IV&V is an effective management tool in developing quality software, caution must be taken in placing too much emphasis on it. IV&V never alleviates the prime contractor's responsibility for mission success, system safety, or other quality assurance practices. To be an effective quality assurance tool, IV&V must complement and reinforce the contractor's software engineering process, configuration management, and qualification test functions [1].

Before discussing the scope and usefulness of an IV&V effort, it will be useful to define some terms.

11.2 VERIFICATION

Verification is Computer Software Configuration Item (CSCI) oriented. As illustrated in Figure 11-1, it is the iterative process of determining whether the product of selected steps of the CSCI development process fulfills the requirements levied by previous steps. Specific task areas that make up the CSCI verification process include:

(a) Systems engineering analytical activities carried out to ensure that the Software Requirements Specification (SRS) reflects the requirements allocated from the System Specification;

(b) Design evaluation activities carried out to ensure that the CSCI design continues to meet the requirements of the SRS as the design proceeds to greater levels of detail (design verification);

(c) Informal testing of the CSCI and its components carried out by the developer to assist in development, provide visibility of progress, and prepare for formal testing (computer program verification);

(d) Formal testing of the CSCI (Formal Qualification Test [FQT]) carried out by the developer in accordance with government approved test plans and procedures to verify that the CSCI fulfills the requirements of the SRS and to provide the basis for CSCI acceptance by the government [1].

11.3 VALIDATION

Validation is system oriented. It comprises those evaluation, integration, and test activities carried out at the system level to ensure that the system that is finally developed satisfies the requirements of the System Specification. Specific validation tasks include:

(a) Systems engineering activities carried out to ensure that the requirements in the system specification accurately respond to the operational needs called for in the Statement of Need (SON);

(b) CSCI integration activities carried out to assemble and check out previously qualified CSCIs as a fully functioning system;

(c) The software aspects of system validation carried out during System Developmental Test & Evaluation (DT&E) and Operational Test and Evaluation (OT&E) to demonstrate that the completed system meets the requirements called for in the system specification (validating the system).

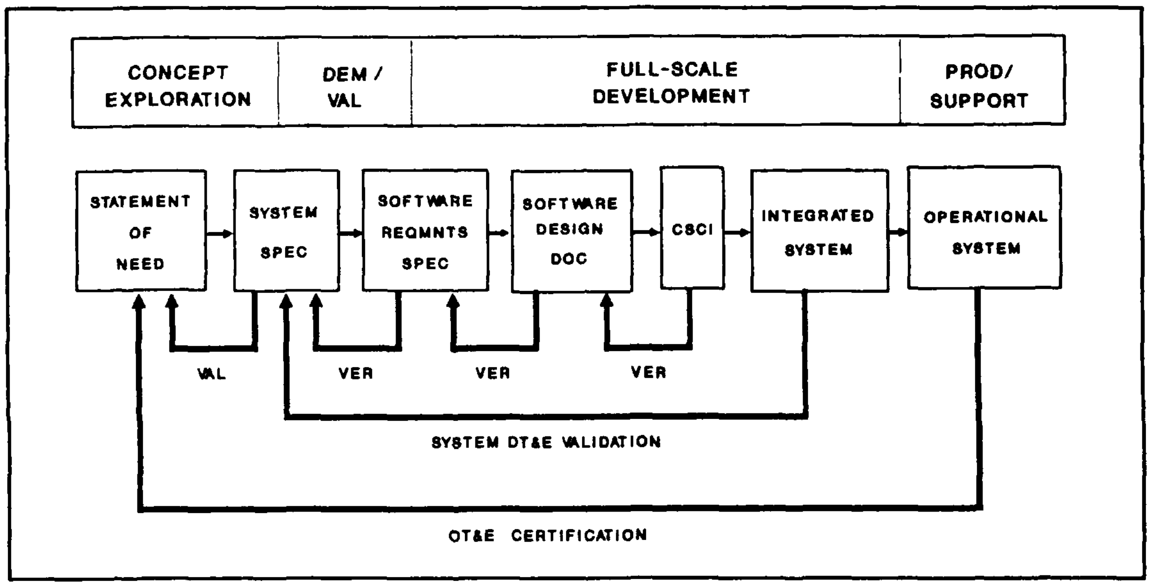


Fig. 11-1 Verification/Validation/Certification

11.4 CERTIFICATION

Certification refers to the using command's agreement, at the conclusion of OT&E, that the acquired system satisfies its intended operational mission. During OT&E the system undergoes test and evaluation aimed at assuring operational effectiveness and suitability under operational conditions [2].

11.5 THE IV&V PROCESS

AFSC/AFLCP 800-5, "Software Independent Verification and Validation (IV&V), Draft" [2] is an excellent source of information on how to:

(a) Determine the need for IV&V

(b) Establish the scope of IV&V

(c) Define the IV&V tasks

(d) Estimate software IV&V cost

(e) Select IV&V agent.

Since this document already addresses these issues, this section will briefly summarize the contents of AFSC/AFLCP 800-5.

11.5.1 Determining the Need for IV&V

Before establishing a software IV&V program it is necessary to determine whether the system being considered warrants an IV&V effort. This determination should be made during the system Demonstration and Validation phase by identifying and examining software requirements. The identification and examination of requirements should be a by-product of the System/Segment Specification (SSS) and the preliminary Software Requirements Specification (SRS). The requirements within each CSCI should be assessed to determine if an undetected error has the potential for causing death or personnel injury, mission failure, or catastrophic equipment loss or damage. A determination should be made to see if the software development effort is to be considered medium to high risk due to technical reasons (i.e., complexity, state-of-the art, system integration, maturity of tools). The CSCIs that meet any of these criteria should be supplemented with an IV&V effort. In addition, the software safety analysis performed as specified in MIL-STD-882 may also determine the need for IV&V.

11.5.2 Establishing the Scope of IV&V

The IV&V effort should be tailored so that it is commensurate with the level of criticality of the software being developed. After determining the need for IV&V, its scope can be established by performing a criticality analysis of the software. This analysis should be done at the CSCI level using the SRS and Interface Requirements Specification (IRS). In many cases, this level of detail is not always available prior to the FSD contract. In these cases, the analysis can be performed at the SSS level. Unfortunately, this may lead to an overall IV&V effort that is too detailed for some CSCIs or not detailed enough for others. The program manager can partially deal with this problem by placing the IV&V agent on a level of effort type contract until the effort can be better defined.

Regardless of whether one uses the SSS or SRS, the criticality of software requirements is performed using the following six-step procedure (See AFSC/AFLCP 800-5 for detailed information):

(a) List the software requirements;

(b) Identify the potential impact of undetected software errors or faults on the software requirements ;

(c) Estimate the probability of occurrence for each error;

(d) Calculate the criticality value of the requirements as shown in Table 11-1 (See ASFC/AFLCP 800-5 for details);

Probability of Occurrence	Impact Categories			
	Catastrophic	Critical	Marginal	Negligible
Frequent	12	9	6	3
Probable	8	6	4	2
Improbable	4	3	2	1
Impossible	0	0	0	0

Table 11-1 Requirement Criticality Values

(e) Calculate the overall criticality value for the system or for each CSCI.

(f) Select the appropriate level of IV&V based on system's/CSCI's criticality value.

11.5.3 Defining the IV&V Tasks

Once the criticality values are calculated the appropriate level of IV&V can be selected as shown below:

Criticality Value	Appropriate IV&V Level
6-12	I
3-6	II
2-3	III
0-2	None-III

Where the three levels of IV&V tasks include various tasks that grow in scope and complexity as you progress from one level to the next. The IV&V levels are defined as:

<u>Level III</u>

(a) Evaluation of software documentation;

(b) Participation in milestone reviews and formal qualification testing, evaluate test plans;

(c) Identification of critical requirements and design issues;

(d) Monitoring the development process and providing technical consultation;

(e) Evaluation of critical test results;

(f) Performing selected audits.

<u>Level II</u>

Same tasks as Level III plus the following:

(a) Analysis of selected critical functions;

(b) Spot checking design performance;

(c) Independently testing critical code;

(d) Analysis of developer's test results;

(e) Independent evaluation of all software problem reports.

<u>Level I</u>

Same tasks as level II plus the following:

(a) Structural and functional analysis of requirements, design, and code;

(b) Propose alternative designs for critical design areas;

(c) Redevelop key code;

(d) Conduct special test in critical areas beyond contractor's formal qualification tests (i.e., stress testing, simulations).

Each IV&V level is progressively more detailed and comprehensive than the previous level. The realism of the criticality values depend on the experience level and background of the people performing the analysis. To select the scope of IV&V and to ensure that its results are not biased, someone other than the software developer or potential IV&V agent should perform the analysis.

It is important to note that IV&V involves activities over the entire software life cycle and that it is more than just another software test activity. Too often the majority of the IV&V effort doesn't start until the software is in the test phase when problems become more visible. Waiting until testing to identify and remove problems loses most of the benefits associated with IV&V.

The tasks performed by the IV&V agent can vary with respect to costs versus benefit. The program office should make provisions to terminate an IV&V task when its costs exceeds its benefit. The program office should establish criteria and/or thresholds for terminating IV&V support. If the IV&V agent is the supporting command for example, program management responsibility transfer might be chosen as the termination point. If the IV&V contractor is for level III tasks, then completion of reviews on the operations and maintenance manuals may be the termination criteria. A termination clause for an IV&V effort should be included in all IV&V contracts or tasking documents.

11.5.4 Estimating Software IV&V Costs

There are no magic formulas to help predict IV&V costs. IV&V cost estimating requires a thorough understanding of the system software and sound judgement. This section provides some simple guidelines to follow when estimating IV&V costs. These guidelines are based on historical trend data and past experience.

In general, the cost of software IV&V can range from 10 to 50 percent of the cost of developing the software depending on the IV&V level selected [1]. The majority of IV&V programs are usually at the lower end of this cost range. Figure 11-2 graphically depicts the relationship between the IV&V level and cost as a percent of software development cost. From this figure and the software's criticality value, the IV&V cost as a percentage of the total software or CSCI's development cost can be estimated. These estimates may be on the high side because they were taken from programs that did not use DOD-STD-2167A, "Defense System Software Development" or MIL-STD-1815A, "Ada Programming Language". Use of these two standards should decrease IV&V costs because both standards increase the engineering discipline in the software development process which improves software quality and makes the IV&V effort easier.

The above guidelines assume that the software development cost has been correctly estimated and accurately defined in terms of:

(a) Size (lines of code);

(b) Complexity;

(c) Volatility of requirements;

(d) Programming language;

(e) Quality of existing software and documentation (when modifying or using existing software);

(g) Maturity of system software and development tools.

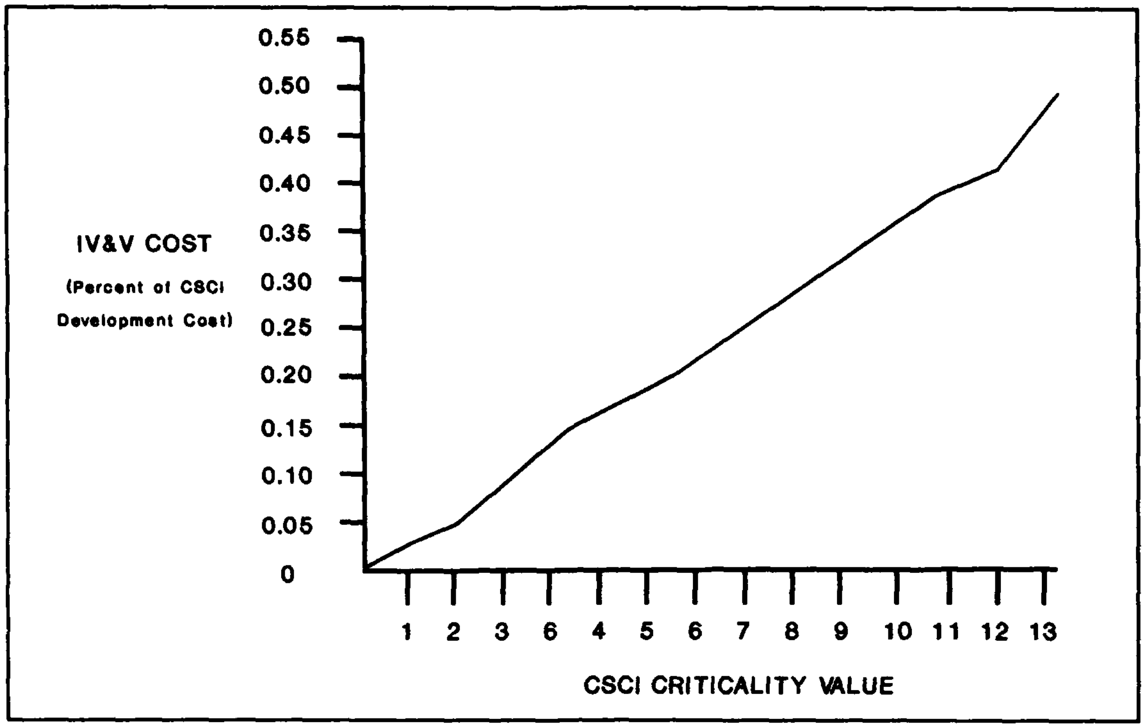


Fig. 11-2 IV&V Cost Vs. Criticality Values

Variations in any of these factors can impact software development cost and subsequently IV&V cost. Before attempting to estimate IV&V cost, it is very important to have a thorough understanding of the software requirements.

IV&V costs should seldom, if ever, exceed 50 percent of the software development cost. Above the 50 percent level, it means that one expects software failures to have catastrophic effects on the program and that such failures are highly likely to occur. If this is the case, it probably means that the application is beyond the state-of-the-art or at the very least on the leading edge of technology and may not be ready to proceed into full scale development.

11.5.5 Selecting IV&V Agent

Selecting an IV&V agent is always the responsibility of the program office. The IV&V agent selected must be autonomous from the software developer. The IV&V agent may be a part of the prime's organization but must report to a level above the head of the software development team (similar to any contractor's quality assurance organization). However, there can be significant benefits from using the software support activity (SSA) as the IV&V agent. This gives the SSA a vested interest in the software development since they must support the system after system delivery.

The following guidelines should help the program office in selecting the best qualified IV&V source:

(a) IV&V Experience. IV&V involves the use of specialized techniques performed by highly skilled personnel for detecting errors not usually found by the software developer. Therefore, one of the most important qualifications of a potential IV&V agent is experience. The IV&V agent must have a strong background in the latest IV&V techniques and have a successful track record. A large amount of IV&V experience gives the IV&V agent the intuitive insight into where problems most likely occur; the ability to quickly recognize problem areas and pit falls; expeditiously recommend sound solutions; and experience in working harmoniously with the software developer in a pressure filled environment. It is imperative that the agent's IV&V experience be commensurate with the level of IV&V desired on that application. One does not want an IV&V agent working on a level I IV&V effort whose entire experience is limited to level III IV&V.

(b) Application Experience. The IV&V agent must have experience in developing or performing IV&V on similar type systems and be qualified in the technology area under development;

(c) Personnel Experience. The management and technical personnel must have a strong background in conducting IV&V on similar programs and be proficient in using their company's IV&V tools. In addition, they must also have detailed knowledge of similar systems and the technology areas being developed;

(d) IV&V Tool Library. The IV&V agent should have IV&V tools that are appropriate for the specified IV&V level. These tools should have been regularly used in the past and not require inordinately specialized talent to use. The existence of a large tool library is of little value if the tools cannot be easily used or the IV&V agent is not thoroughly familiar with their use and limitations.

To determine the qualifications of an IV&V agent the program manager has two options. The first is to review the agent's past performance on similar applications. The second is to perform a software capability/capacity review on the IV&V agent as described in Chapter 8.

11.6 REFERENCES

1. AFSC/AFLCP 800-5, "Software Independent Verification and Validation (IV&V)", 1988.

2. ESD-TR-77-326, "Software Acquisition Management Guidebook: Validation and Certification", Electronic Systems Division, Hanscomb AFB, MA, August 1977.

CHAPTER 12
METRICS

12.1 INTRODUCTION

This chapter deals with the mechanics of measuring the progress of a software development effort. The material is addressed from a top level perspective and is typical of the data that is usually presented to a program manager. Because the data will be summary level data, additional data at the next level of detail may be required to support a thorough review. The software manager should have more details and ready access to the contractor's data. To receive metrics data requires that this requirement be clearly called out in the contract. It is important to examine the contractor's process to determine what data is available and to use the contractor's system rather than to impose unnecessary requirements such as peculiar formats. If the data is not available, it may indicate a weakness in the contractor's ability to provide visibility to the program office.

There are many different software metrics available but few of them are proven. The state-of-practice in software metrics is far from perfect, but there are a few recommended metrics that are appropriate for all programs. Metrics are generally relevant only to an individual program and indicate trends or alert flags; they are not absolutes. In other words, a specific value may not describe absolute performance or cost but the collective data may provide a level of confidence on the range of performance or cost.

The following material is extracted almost entirely from the Air Force Systems Command, Electronics Systems Division and MITRE document, ESD-TR-88-001, "Software Management Metrics".

12.2 PROGRAM MANAGER'S METRICS

Table 12-1 lists the metrics that should normally be available, at least monthly, to the program manager. The basic method requires an initial planned estimate as a baseline to measure progress against. Actuals are then tracked against the baseline plan and trends or thresholds are analyzed for progress or problems.

Although there are many more metrics available, the list in Table 12-1 is an attempt to provide the program manager with a "Top Ten List".

* Software Size and Cost Status
* Manpower Application Status
* Cost and Schedule Status
* Resource Margins
* Quantitative Software Specification Status
* Design and Development Status
* Defects/Faults/Errors/Fixes
* Software Problem Report Status
* Test Program Status
* Delivery Status

Table 12-1 Metrics

12.2.1 Software Size and Cost Status

The software size metric tracks the magnitude of the software development effort while the cost metric tracks expenditure of resources (Figure 12-1). The metrics for each Computer Software Configuration Item (CSCI) are normally tracked individually. The trend should be fairly stable, although a deviation may indicate a better understanding of the requirements. One should look for an unstable baseline, requirements growth, and poor planning. The software size metric is generally the input to a cost model such as COCOMO [1]. Changes in size requires more resources and reflect a change in cost. The cost model is not shown.

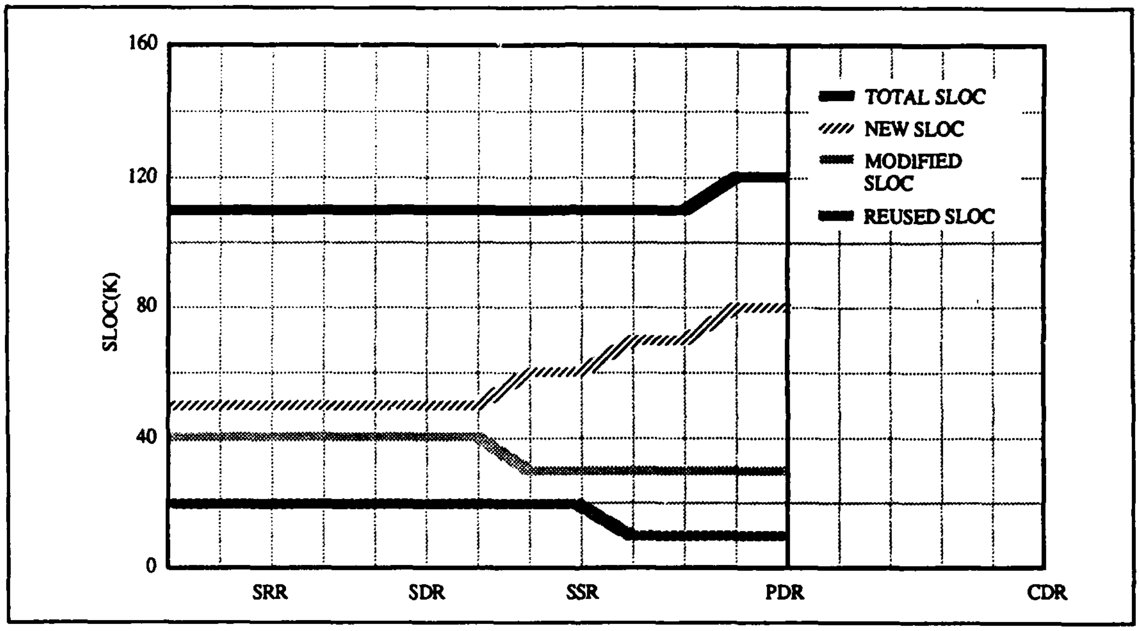


Fig. 12-1 Software Size

Rules of Thumb for Software Size and Cost Metrics

(a) Estimates should not vary by more than 5% from month to month. Variation may indicate a better understanding of the requirements or problems with the contractor's development process.

(b) Typical effective weights for source lines of code (SLOC) should be: New code 100%, Modified code 50% , Existing code 10%.

(c) SLOC requirements directly impacts manpower requirements.

(d) Use higher order languages.

(e) Tailor the metrics by tracking:

- Equivalent SLOC (weighted);

- Each programming language;
- The metrics for each CSCI separately;
- Object code size.

12.2.2 Manpower Application Status

Tracking planned versus actual manpower loading provides visibility into future schedule problems (Figure 12-2). Resources should be identified as either experienced/senior personnel or inexperienced/junior personnel. Tracking losses is also important, particularly if the losses are in key positions such as the chief programmer.

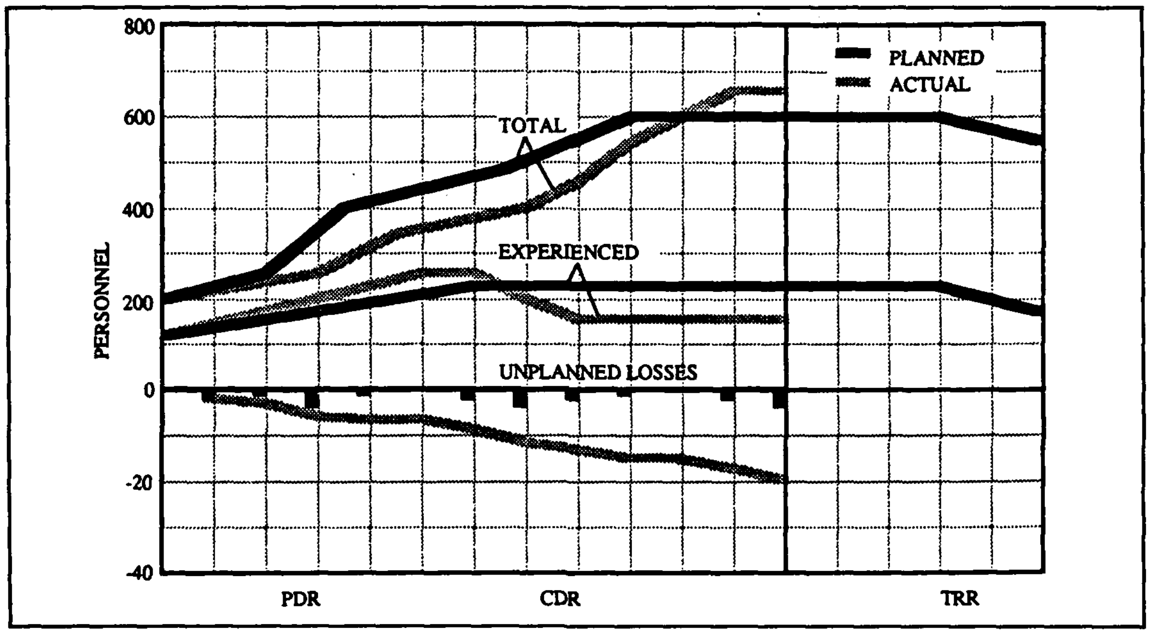


Fig. 12-2 Software Personnel

Rules of Thumb for Manpower Metrics

(a) The ratio of total to experienced personnel should never exceed 6:1. A ratio of 3:1 is typical.

(b) Initial staffing usually comprises about 25% of total personnel requirements.

(c) The front end should be leveraged with more experienced personnel.

(d) The development schedule depends on the amount of man-months expended:

- Understaffing is an early sign of schedule slippage;
- If you are behind, you can't always catch up by adding more manpower. Adding manpower may even further delay the schedule;
- Judicious use of overtime may help;

- Additional use of manpower may work but only for tasks that can be separated or isolated;

(e) High turnover or loss of key personnel is a sign of problems;

(f) Tailor the metrics by tracking the staffing for each:

- Development task;
- Skill (e.g. Ada, Data Base Management Systems, Artificial Intelligence);
- Organization (e.g. Software, Quality Assurance, Test).

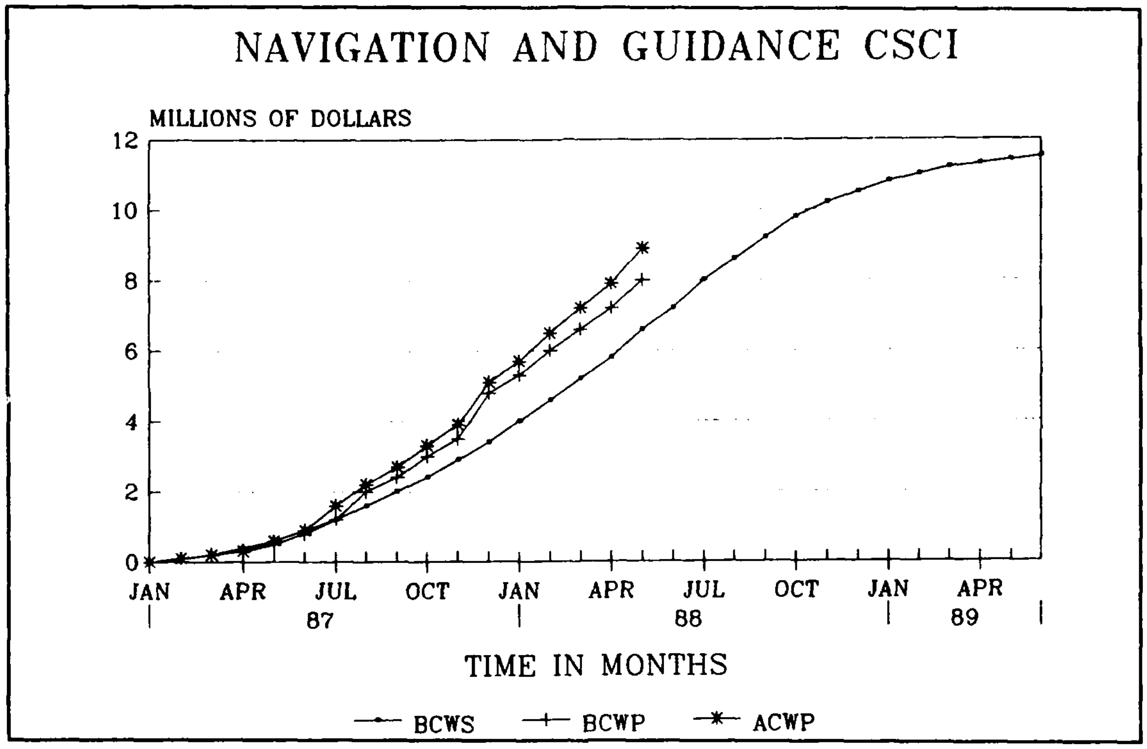


Fig. 12-3 Software Cost and Schedule

12.2.3 Cost and Schedule Status

Financial reporting provides the status of work performed and actual cost of work performed versus the plan (Figure 12-3). This is the traditional Cost/Schedule and Control System. Typically, the data reported to the government provides little visibility into the software development status. This usually occurs because of inadequate definition of the software WBS as well as the level of reporting. The WBS levels must provide reasonable visibility to the program manager. One way to do this is by defining a product oriented Work Breakdown Structure (WBS) for the software.

<u>Rules of Thumb for Cost and Schedule Metrics</u>

(a) Be alert to variation thresholds exceeding 10%.

(b) Beware of efficiencies that are projected to improve. Past performance is a measure of productivity and efficiency and it is very useful for forecasting estimates to complete. Efficiencies don't usually improve that much.

(c) Tailor metrics by tracking:

- Each CSCI;
- Support software development.

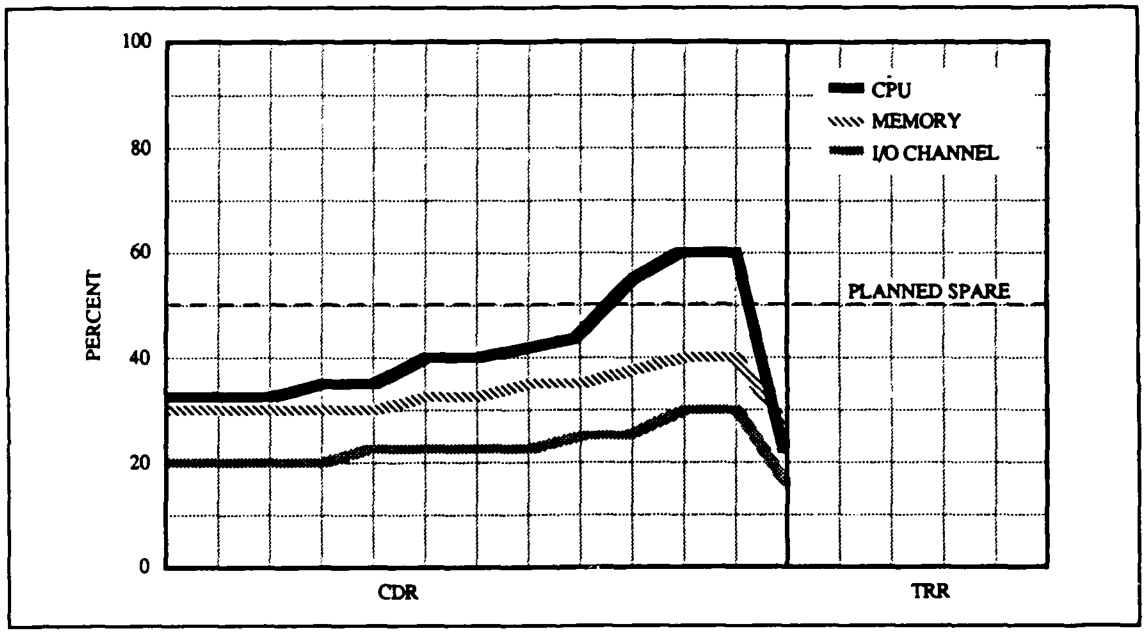


Fig. 12-4 Computer Resource Margins

12.2.4 Resource Margins

Resources describe the hardware limitations of the systems. These include Central Processing Unit (CPU) throughput, memory size, and Input/Output channel capacity and rate (Figure 12-4). Resources can have a direct impact on software productivity and design efficiency. The development resource margins are as important as the target resources particularly for planning software support.

<u>Rules of Thumb for Resource Margin Metrics</u>

(a) CPU utilization should allow for a 50% margin at delivery (this means that only half of the resource has been used);

(b) Memory utilization should allow 50% margin at delivery;

(c) I/O utilization (channels and data rates) should allow 50% margin at delivery;

(d) For real-time systems, performance/productivity deteriorates quickly above 70% utilization;

(e) Consider hardware resource limitations (e.g. memory addressing as a hardware limit);

(f) Resource utilization tends to increase with time so plan for expansion;

(g) Schedule and cost "bomb" at 10% margin (in other words you'll already have seen an exponential rise in the effort required to squeeze, pare, re-code, fix, etc.);

(h) Tailor metrics by tracking:

- According to architecture (e.g. multiple CPUs);
- Average and worst case;
- Host and Target equipment;

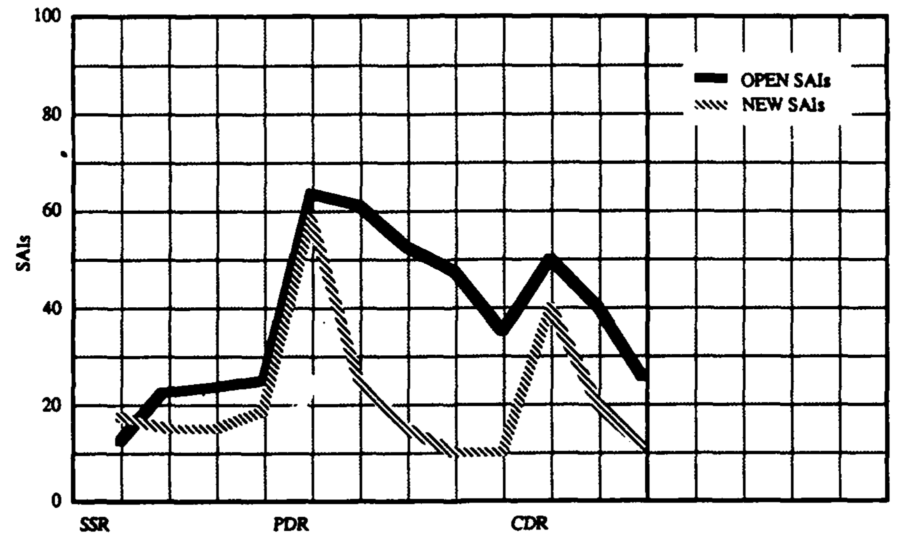


Fig. 12-5 Software Volatility/Action Items

12.2.5 Quantitative Software Specification Status

A baseline plan is initially estimated for the quantity of discrete software requirements to include both functional requirements and interface requirements. Actual progress is tracked against the plan. This metric provides visibility into the progress of the requirements analysis as well as the growth of the requirements in the baseline (Figure 12-5 and 12-6).

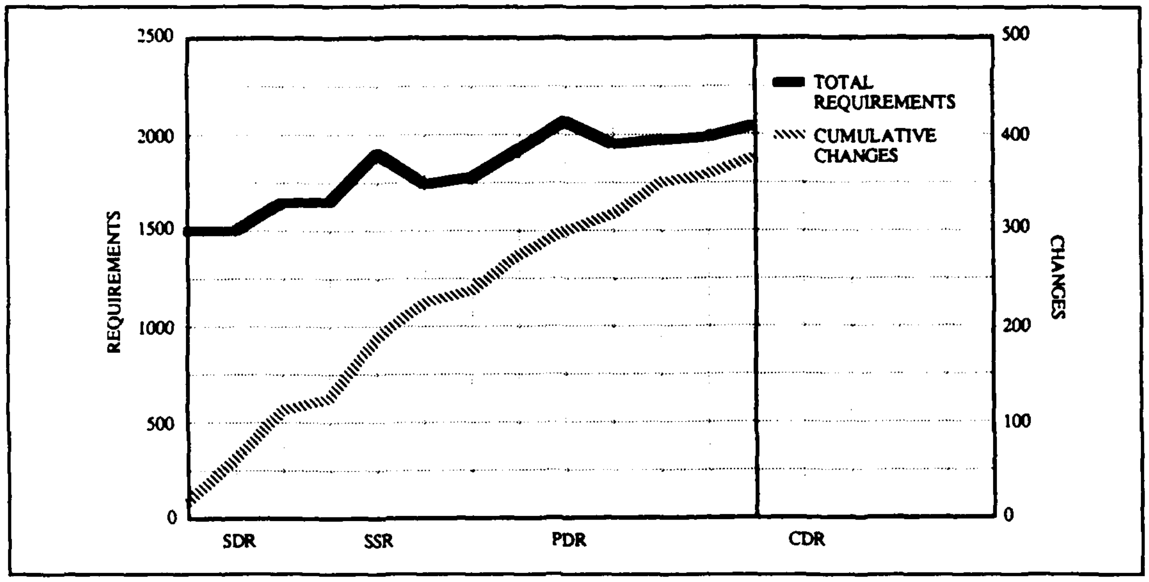


Fig. 12-6 Software Volatility/Requirements Changes

Rules of Thumb for Specification Metrics

(a) Each requirement should have a planned completion date;

(b) Requirements growth, no matter how small, will impact planned resources and should be contained from the beginning;

(c) Requirement uncertainty leads to Engineering Change Proposals (ECPs);

(d) Requirements are baselined at the Software Specification Review (SSR);

(e) If the requirements are not stable by the Critical Design Review (CDR), the program is in serious trouble;

(f) Requirements change after CDR will most probably impact the schedule;

(g) Phase incremental development to allow the requirements to be revisited before the Preliminary Design Review (PDR) of the next increment;

(h) Software action items should not remain open beyond 60 days.

12.2.6 Design/Development Status

The contractor should describe in the Software Development Plan the process for inspections, walkthroughs and internal design reviews. These events can be tracked to determine the

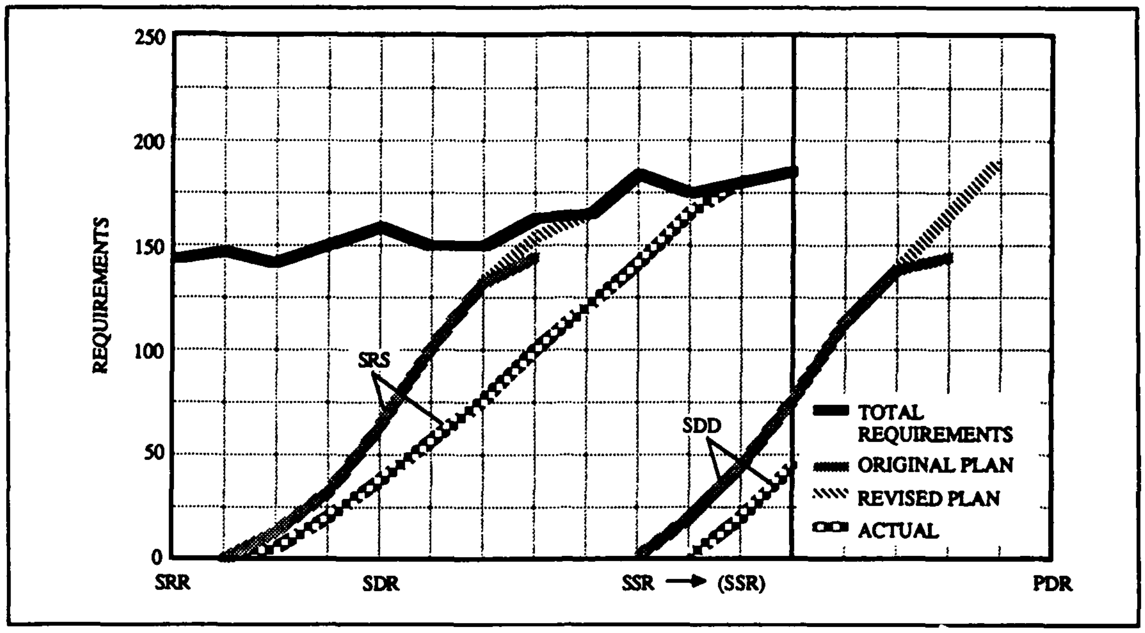


Fig. 12-7 Design Process

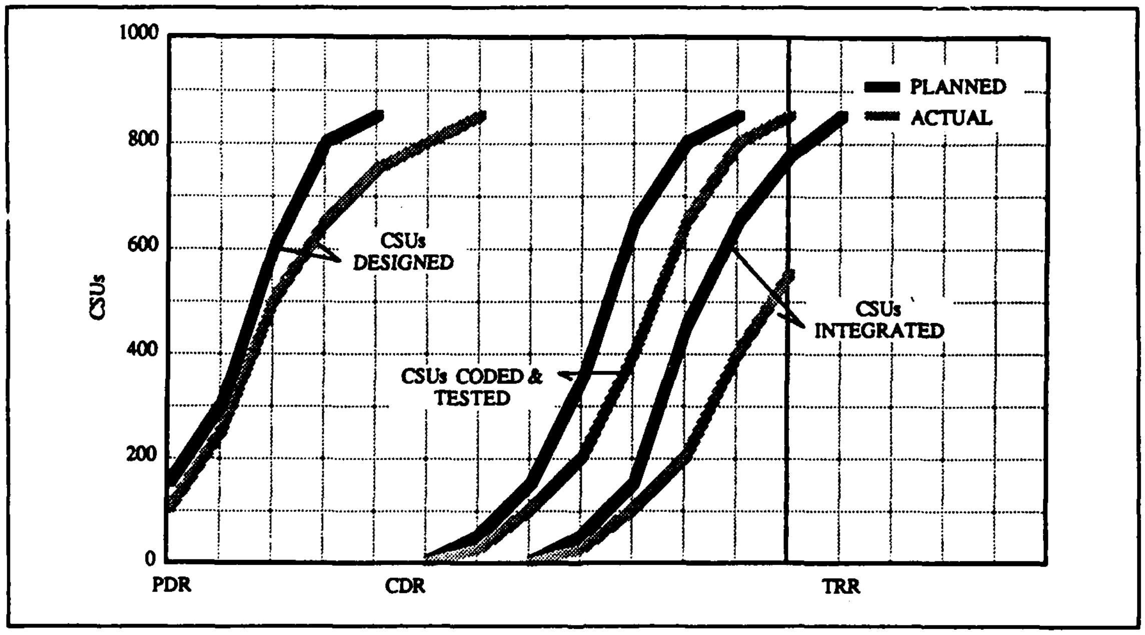


Fig. 12-8 Development Progress

rate of progress. On a large project these events can number in the thousands (Figures 12-7 and 12-8).

<u>Rules of Thumb for Design Metrics</u>

(a) The Software Requirements Specification (SRS) should be complete before the Software Specification Review (SSR);

(b) The Software Design Document (SDD) should be complete before the PDR;

Rules of Thumb for Development Metrics

(a) The CSU design should be complete before the CDR;

(b) CSCI integration and test must be completed before the Test Readiness Review (TRR);

(c) Diverging from plan may mean schedule delays;

(d) Track to cost model (e.g. COCOMO);

(e) Source Lines of Code (SLOC) per staff month can be categorized as:

Easy Code	- 150
Moderate Code	- 70
Difficult Code	- 30

(f) Tailor metrics by tracking:

- Each CSCI;
- Internal reviews of program design languages (PDLs);

(c) Delaying development to obtain a better understanding of the requirements is usually a wise decision. Tradeoffs may be made to reduce requirements for gains in schedule;

(d) Diverging from the plan means that the requirements are less understood. You may not be ready for the SSR;

(e) Tailor the metrics by tracking more detail, from the SDD to the Computer Software Unit (CSU).

12.2.7 Defects/Faults/Errors/Fixes

Tracking the actual performance of the process provides some visibility into the product quality. This is only true if the contractor has a controlled, repeatable process. Ad Hoc software development is not predictable.

A defect is an anomaly in the requirements and design. A fault is an anomaly in implementation (code). An error is the source of a fault and a fix is a correction of an error. Errors and fixes are tracked through the Software Problem Reporting (SPR) system (Figure 12-9)

Rules of Thumb for Defects/Faults/Errors/Fixes/Metrics

(a) Defect/fault/error rates are an early indication of product reliability.

(b) The contractor's process should focus on defect prevention and early fault detection.

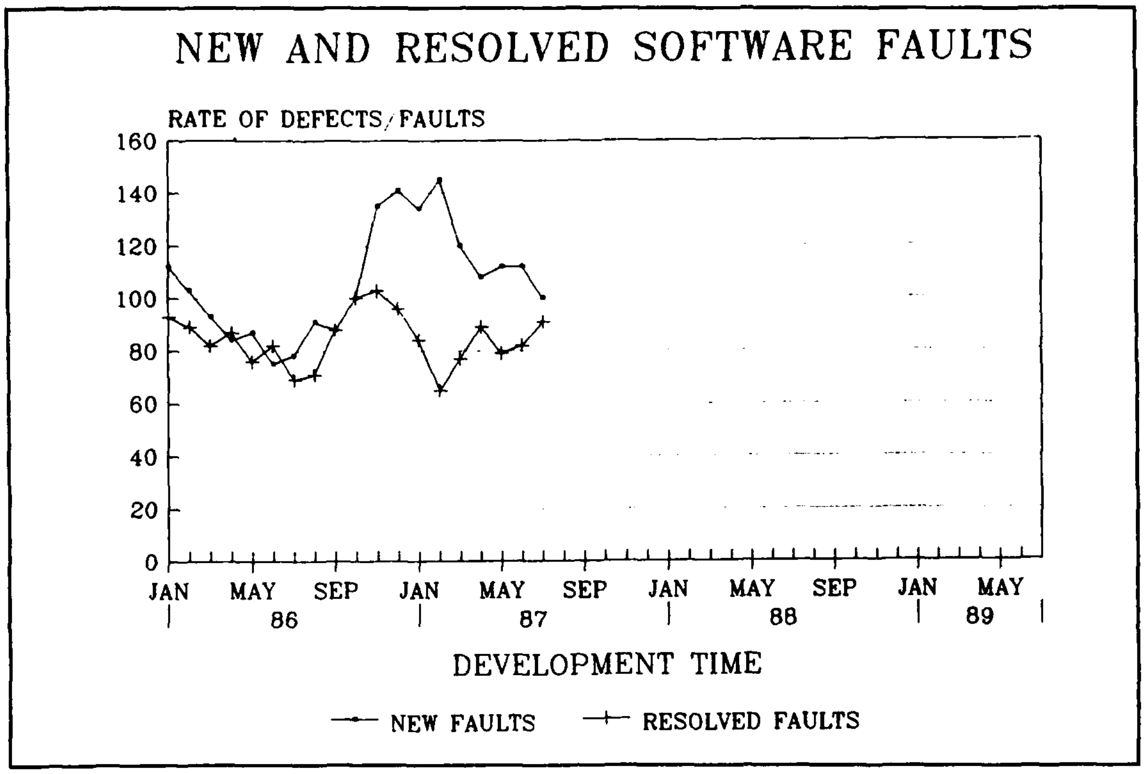


Fig. 12-9 Defects/Faults/Errors/Fixes

12.2.8 Test Program Status

The entire test process can be tracked from the beginning of planning through detection of errors and correction (Figure 12-10). Quantitative measures can start with a requirements and test cross reference matrix. Progress can also be measured on planned versus actual tests performed.

12.2.9 Software Problem Reports Status

Software Problem Reports (SPRs) provide feedback on the correction of errors and are an indication of product quality (Fig. 12-11)

<u>**Rules of Thumb for SPRs**</u>

(a) Planning should account for the iterative and interactive nature of testing;

(b) The number of tests completed should converge on the number of tests planned;

(c) Use trends to predict schedule;

(d) Unresolved problems should decrease to zero as you approach the TRR and again as you approach the Physical Configuration Audit (PCA);

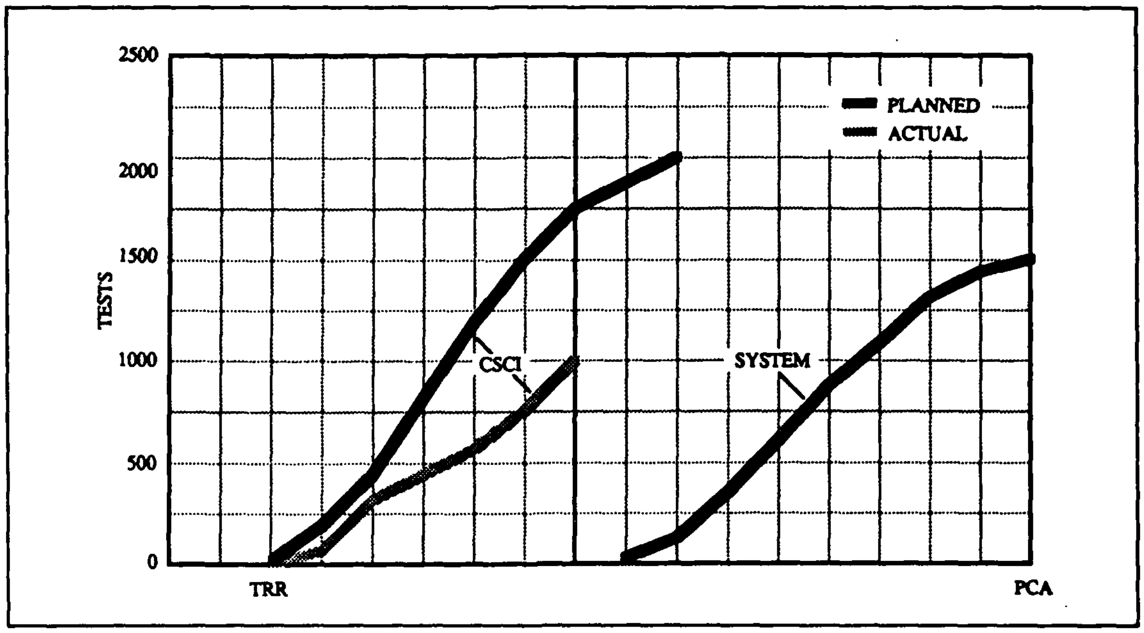


Fig. 12-10 Test Program Status

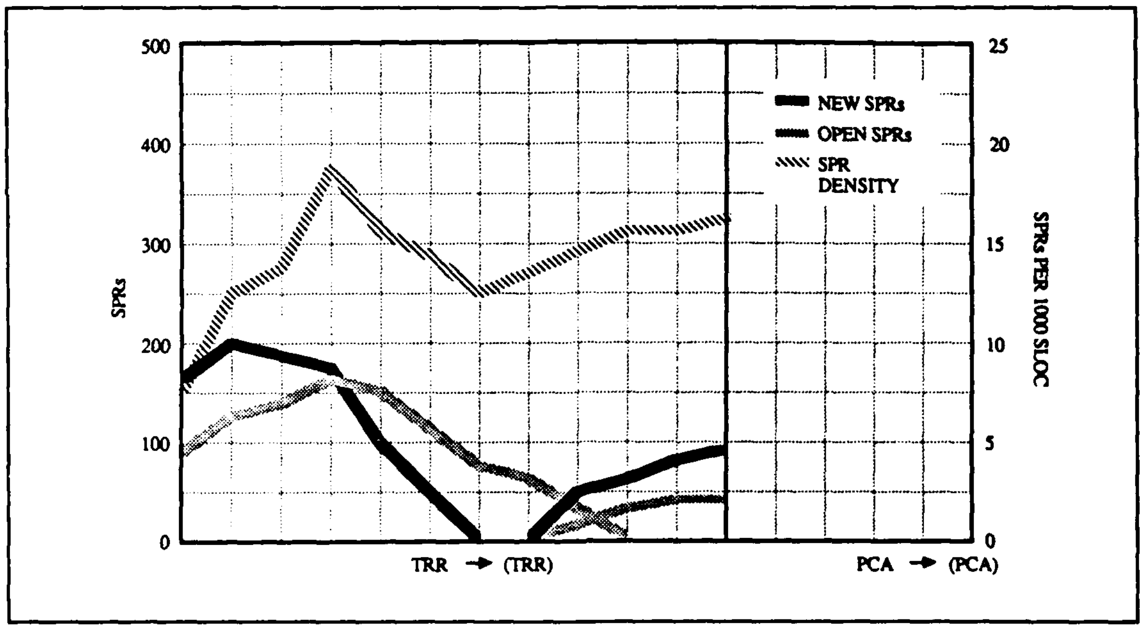


Fig. 12-11 Software Problem Reports

(e) The number of SPRs is an indication of the testing adequacy and the code quality. Too many SPRs may indicate poor quality; too few may mean inadequate testing. It takes a process change to improve the rate and the quality. In today's state of practice a typical range is 5 to 30 SPRs per 1000 SLOC.

(f) If the slope of open SPRs is positive then problems are being found faster than they are being fixed. If the slope is negative then the schedule can be predicted.

(f) If the slope of open SPRs is positive then problems are being found faster than they are being fixed. If the slope is negative then the schedule can be predicted.

(g) Tailor the metrics by tracking:

- The number of days SPRs are open (e.g. 0-30, 30-60, 60-90, over 90);
- Open SPRs by type of software (e.g. application, support, test, operating system);
- Open SPRs by priority (e.g. critical to operation, critical for integration, other);
- SPR density (SPRs per 1000 SLOC in categories 0-10, 11-20, 21-30, and over 30) .

12.2.10 Delivery Status

A good indicator of progress is to track both internal and external incremental delivery status (Figure 12-12). For example, one can track internal delivery to the test organization for integration and test and external delivery to an IV&V agency. Often, early problems in a release may be deferred to a later release. This delay should recognize the shift in resource requirements. Too many deferrals can spell disaster.

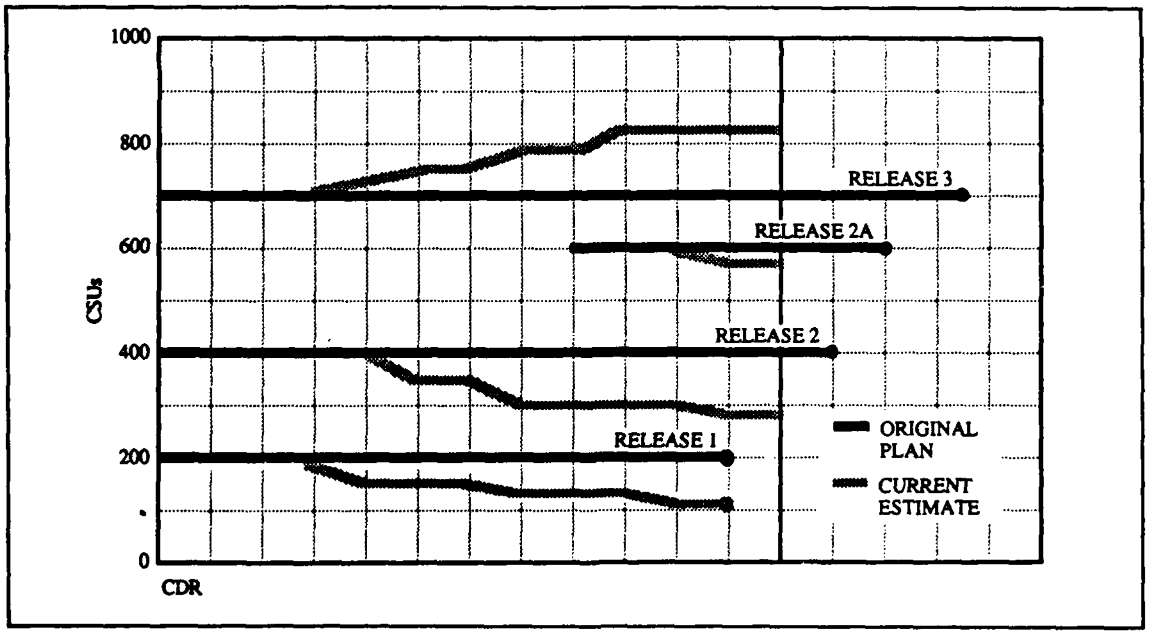


Fig. 12-12 Incremental Release Count

<u>Rules of Thumb for Delivery Status Metrics</u>

(a) The number of CSUs per release should remain stable (within 10%).

(b) Increments or "builds" should demonstrate useful capabilities as early as practical.

12.3 SUMMARY

Metrics should provide the program manager a tool for much greater visibility into the software development than in the past. The metrics discussed in this chapter focused primarily on tracking progress. There are other techniques available to monitor product quality, although these are much harder to quantify. It is important that the contractor have the right mechanisms in place for proper discipline and commitment to quality.

12.4 REFERENCES

1. Boehm, Barry W., Software Engineering Economics, Prentice Hall, 1981.

CHAPTER 13
EPILOGUE

13.1 INTRODUCTION

Improvements in software productivity are coming slowly. Present conventional techniques for software production make use of libraries of primitive functions or algorithms (See Figure 13-1). There is little carryover from past developments. The primary development effort is in-line with

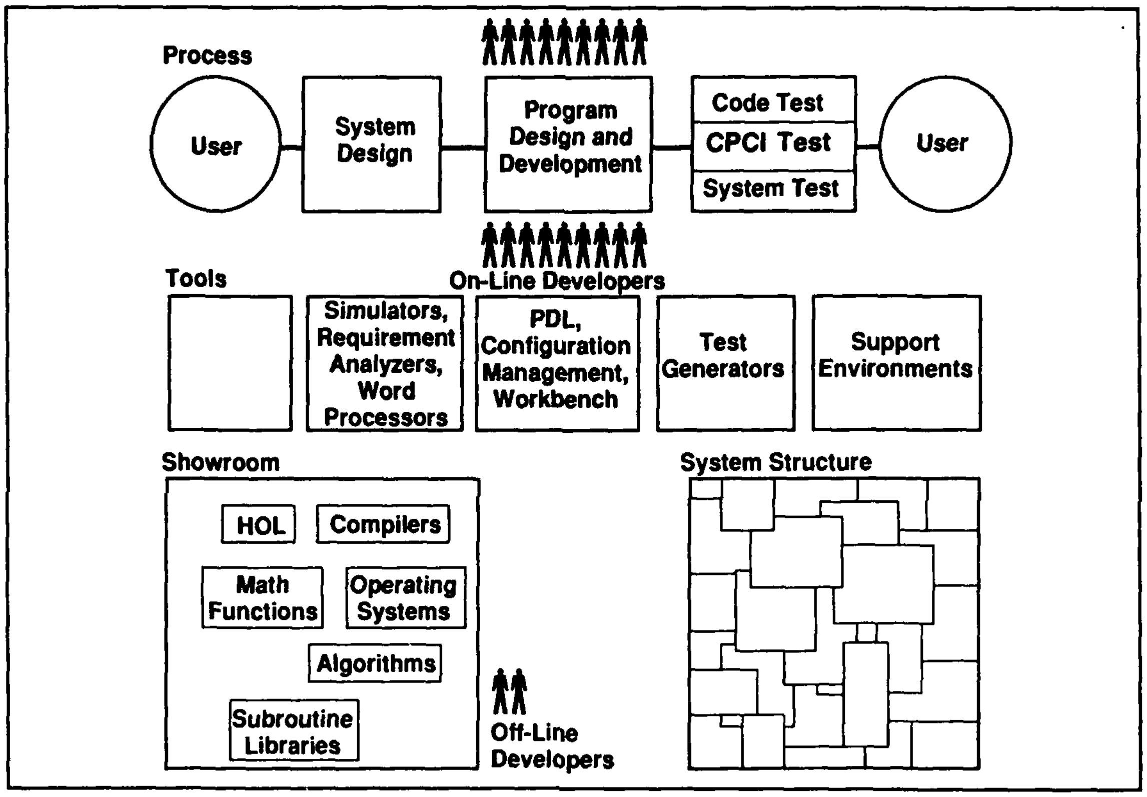


Fig. 13-1 Conventional Techniques

the delivery of the software product or capabilities and is dedicated to a single system. There is little ability to leverage the development investment and provide products for use on the next system. The resulting system structure is typified by many unique modules with complex interfaces requiring significant effort to integrate, test and maintain.

The communications among the various players (user, system designer, programmer) involved in software development, acquisition, delivery and use is constrained. The constraints arise from a number of factors including geography, personnel availability, lack of software development tools, procurement and acquisition policy restrictions, and the diverse backgrounds and training of the players. The resulting situation might be characterized as each player tossing requirements, designs, and comments over a wall to one another.

These constrained communications, and the unique and customized implementations which result from our present ways of building software, usually require that the computer programmers play a key role in the delivery and transition of the system to the user. In fact, in many systems even the system designers and the system test group have a difficult time discovering critical failure modes and correcting problems economically when they are discovered. The need to use the software development organization during transition is often underestimated; especially when the software development organization is not the prime contractor. The prime usually phases out these programmers and performs the final system test and operational test and evaluation without them. Unfortunately, current software development practice makes it difficult for the prime to perform these functions adequately.

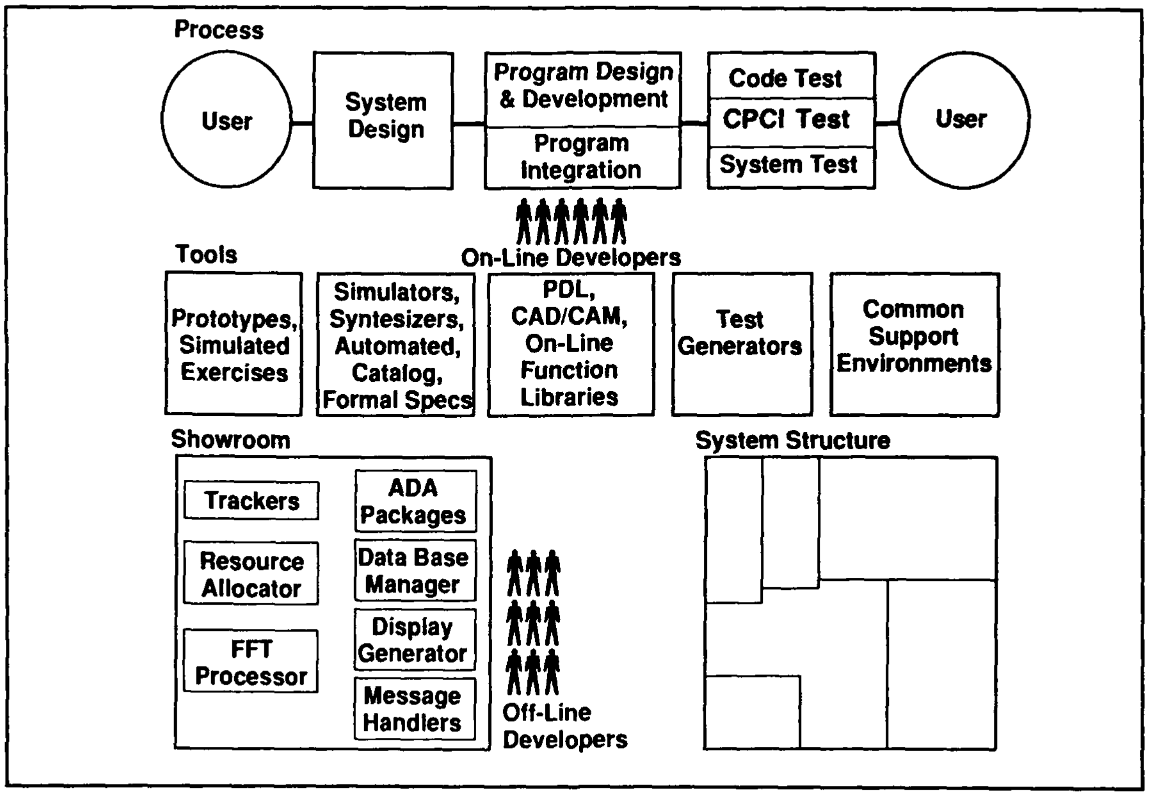


Fig. 13-2 Reusability

There may be another way to build software and overcome some of the drawbacks of our present techniques. A model of an alternate software development technique is shown in Figure 13-2. In this model, showrooms of larger more capable piecesare developed off-line for later integration and use in multiple systems. The in-line activities are, therefore, more heavily directed towards program integration than to design and development. The testing emphasis is more on system testing than code or CPCI testing. Such an approach may support the use of computer aided design and manufacturing (CAD/CAM) for software. The resulting system structure is more regular, has simpler interfaces and is easier to test. Less in-line development is required for the delivery of a software product or capabilities. This allows the off-line development effort to support a larger number of systems.

However, reusing software on multiple systems is a tough job. How can the performance and interface of the modules be described? How well can they be tested? What is their run time? How are they catalogued? Who owns the data rights? How are they guaranteed?

The technology and the management infrastructure needed to fully exploit reusability is not yet available. Efforts in the STARS program and at the Software Engineering Institute are investigating the technical and management advances required to answer these questions and to bring reusability closer to practice. In the meantime, greater use of best existing practice, tailored acquisition strategies, and improvement of tools and techniques are required to help manage the software acquisition process.

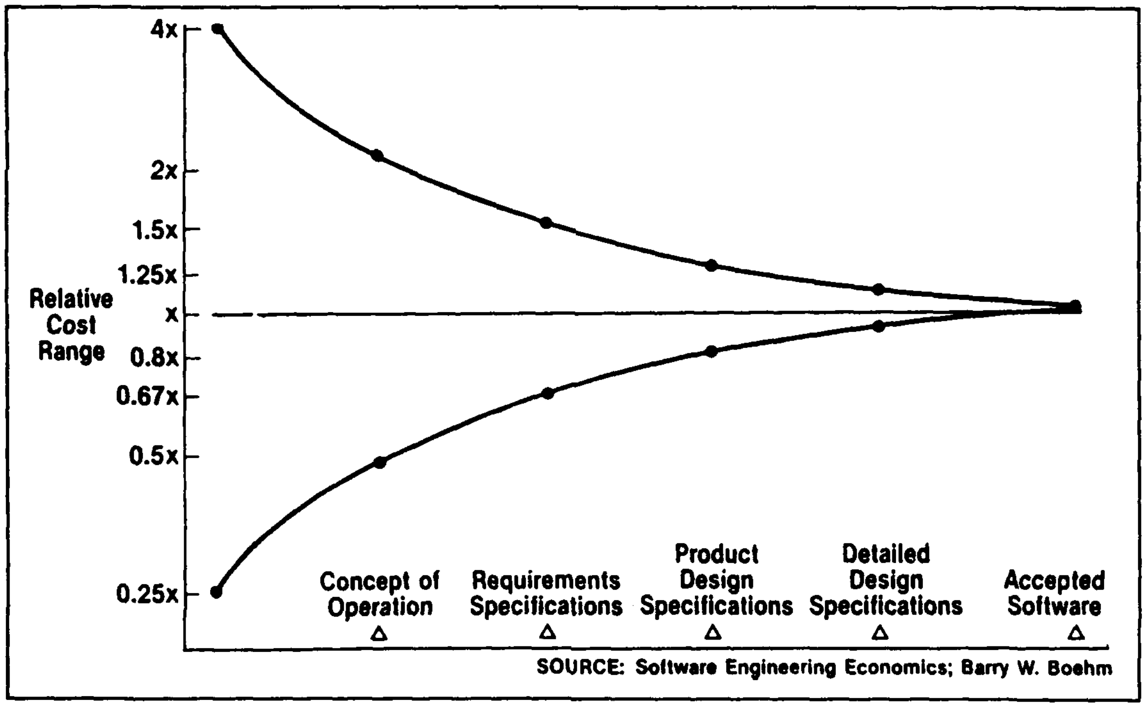


Fig. 13-3 Software Cost Estimating

13.2 SOFTWARE COST UNCERTAINTIES

Because the current nature of the software development process is more like model shop or custom tailoring than like mass production, software cost estimating accuracy is a very strong function of the program phase. The uncertainty in the software development process is captured in Figure 13-3 [1]. Errors on the order of 4 to 1 are likely when estimating software costs at the start of a project.

These early cost estimates require difficult judgments of complexity, productivity and size. Experience indicates that the curve should be one-sided because software costs are never overestimated; they are always underestimated.

How is it then that most cost estimators advertise techniques which purport to provide 10 to 20% accuracy? Most of these estimates make use of regression analyses and deal with the process from the perspective of the right end of the curve. With present techniques, if you want better accuracy, you have to do something to move toward the greater experience part of the curve.

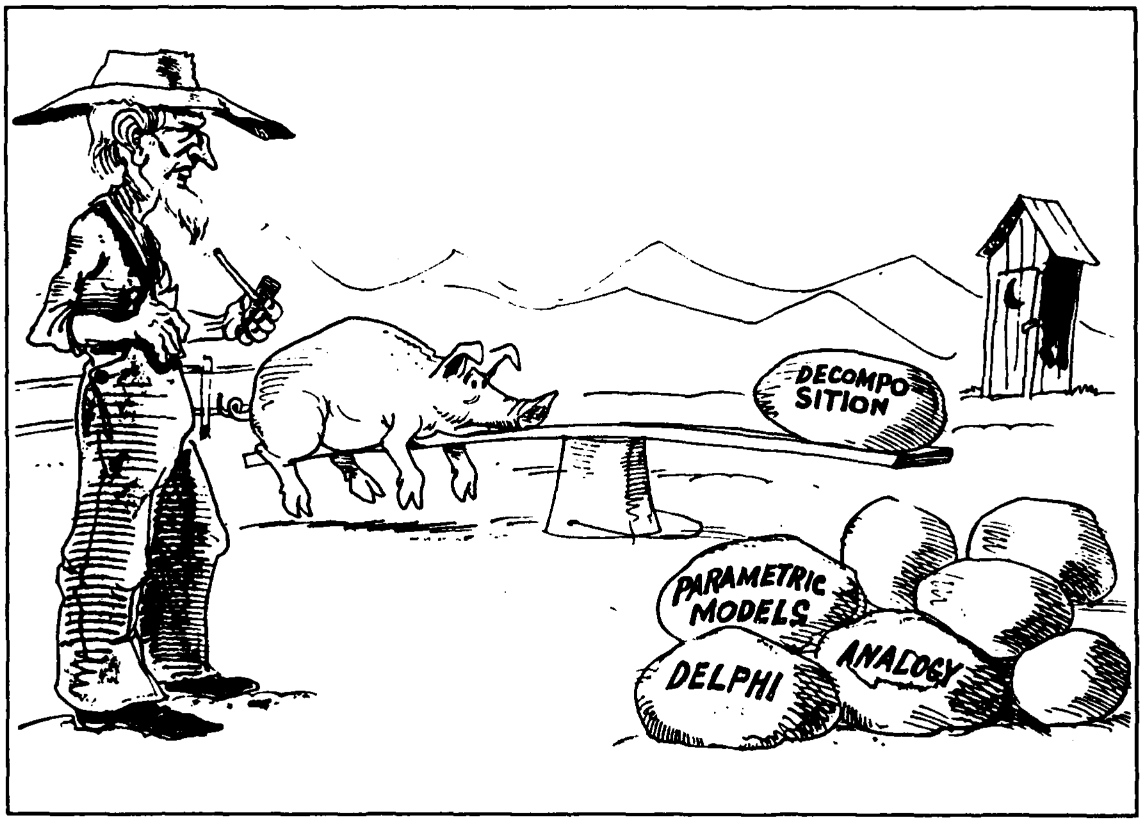


Fig. 13-4 Estimating Techniques

This can be done by building prototypes, making use of previously designed and developed products, using commercial off-the-shelf software, or finding an acquisition strategy which includes a contract definition phase or incremental development. A strategy is needed to delay major fund commitments until better software requirements and implementation definitions are obtained.

Early software cost estimating may be likened to the story of the old farmer who had a unique way of estimating the weight of pigs (See Figure 13-4). "I don't need any of these newfangled scales to do that," he said. He had an easy way to weigh pigs without scales. He laid a plank across a pail, put a stone on one end, a pig on the other, and balanced the plank. Then he guessed the weight of the stone. From that he could easily calculate the weight of the pig --to three significant figures. With current model shop techniques, one cannot write better specifications or obtain better cost estimates without first doing some part of the job to be specified or estimated.

13.3 SOFTWARE ACQUISITION CYCLE

The ideal acquisition cycle includes a concept exploration phase, a demonstration/validation phase, a full scale engineering development phase, a production phase and a deployment phase. In most Command, Control and Communications (C^3) systems acquisitions and in many non-developmental item (NDI) acquisitions, the normal weapon life cycle is compressed (See Figure 13-5) [2].

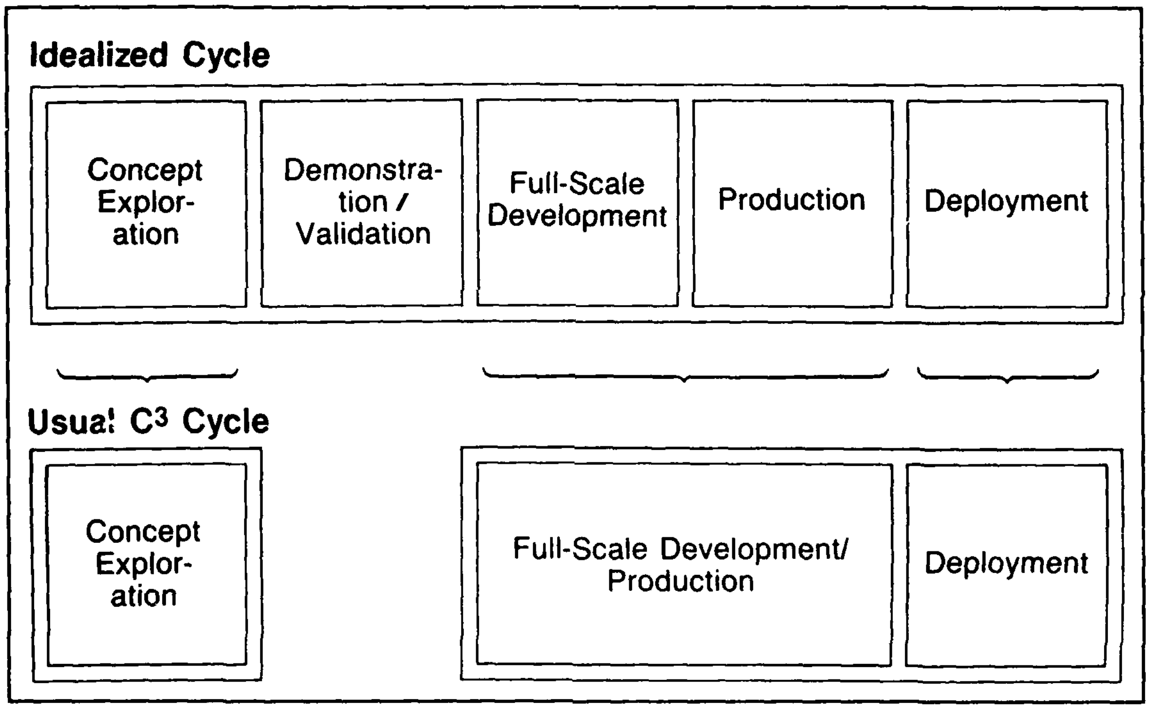


Fig. 13-5 System Acquisition Cycle

Although a concept exploration phase may take place, the demonstration/validation phase is generally omitted and the full scale development and production phases are combined. When no demonstration/validation phase is present, time somehow must be allowed for definition and detailed design. Many acquisition schedules call for preliminary design reviews as soon as three months after contract award. The contractor is forced to either accomplish the top level design in the proposal phase or in the short time available after award.

The use of top-down design has been recommended as an orderly and disciplined way to develop and test software. For new programs, however, one cannot perform top down design and development without some knowledge of the bottom (See Figure 13-6). One must have some assurance that allocations of requirements, from one baseline to the next, will still be valid when the lower level designs are developed. Historically, most Air Defense programs have run out of processing resources as a result of a poor understanding of the need to carefully sort tracks before attempting radar data correlation. Prototypes or breadboards of the difficult parts of the design help validate requirement allocations and avoid breakage and rework.

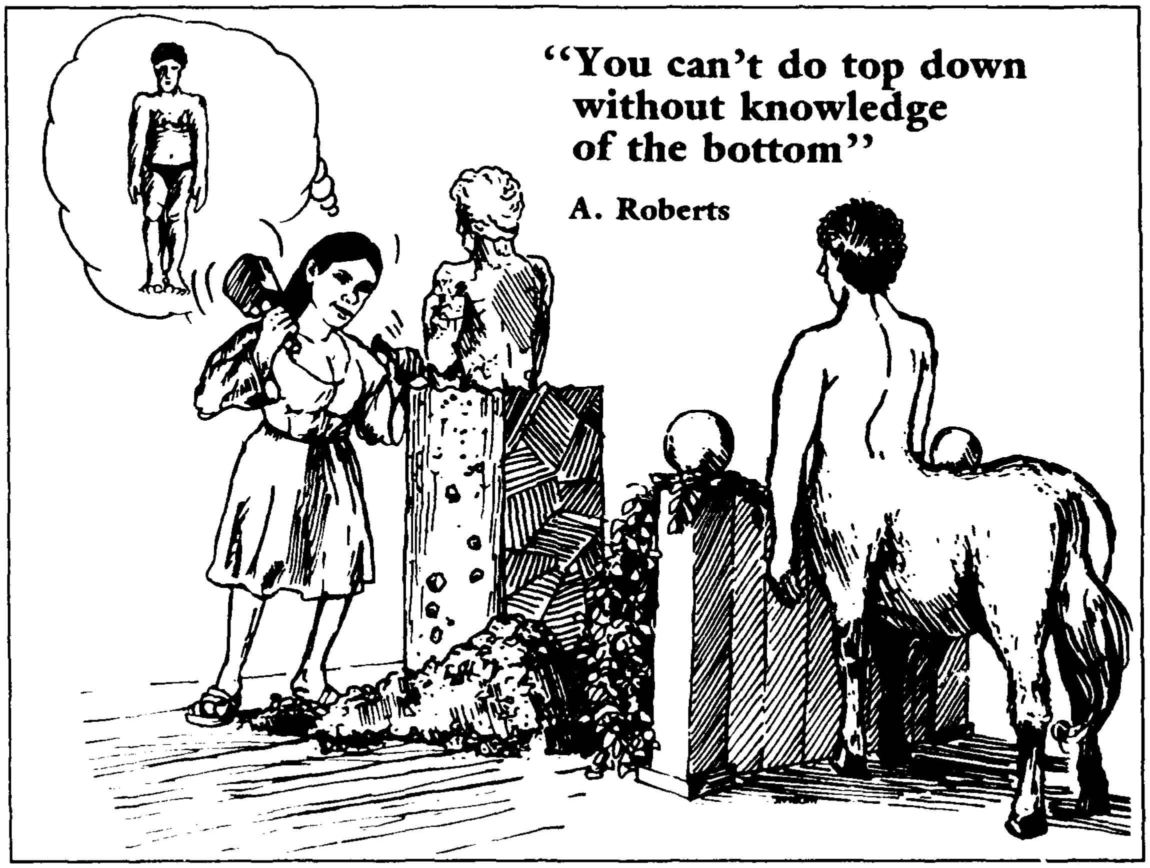


Fig. 13-6 Top-Down Design

13.4 PROTOTYPES

Prototypes are useful in two ways. They help resolve requirements uncertainties by providing better user insight and lessening the misunderstanding that arises from just looking at paper requirements and specifications. They also help reveal implementation difficulties or constraints.

Prototypes generally make use of special operating procedures and operating systems and have limited interfaces and loading. They have no provisions for startover, continuity of operation, or maintenance, and use computers and code that are not intended for the target system. Too often the government uses the information from the prototype to generate high level system performance specifications for competitive bid with a new set of contractors. Much of the experience gained from the prototype development is lost in the process if the prototype developer is not the winner. With today's state-of-the-art software production techniques, it is very difficult to convert a developer's experience into specifications of sufficient quality to allow someone else to gain that knowledge and experience. Unless the original players, both user and development contractor, remain the same and the early prototype is very close to the end item, a second prototype must be built. Otherwise, some strategy such as incremental development must be used to help manage the requirements and implementation uncertainties.

13.5 SCHEDULES AND MANNING

The amount of time the government allows for design, and the rate at which the contractor staffs the software development activity, have a major impact on the success of

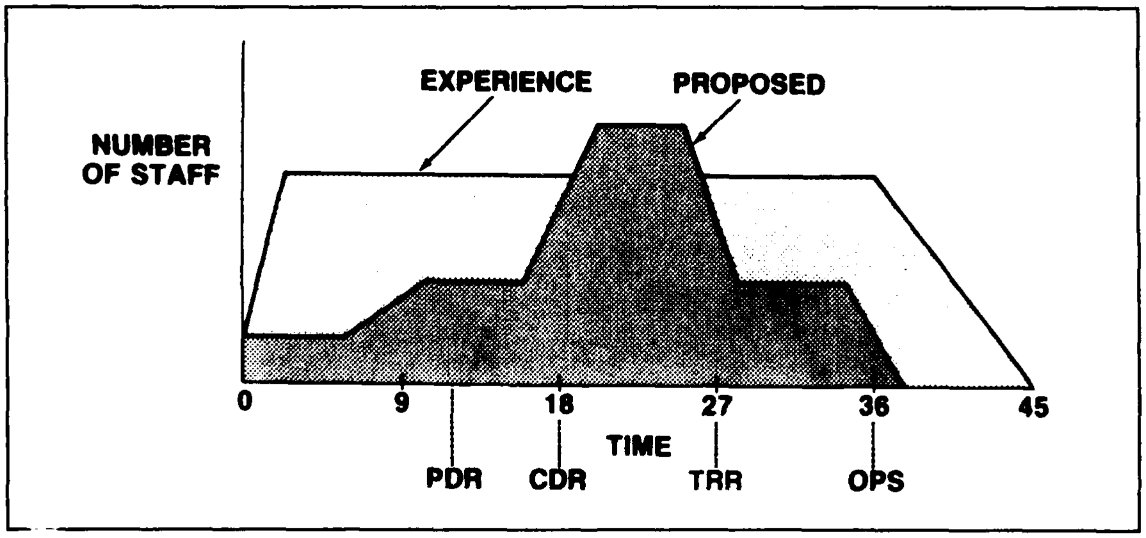


Fig. 13-7 Software Manpower Phasing

the program. A thesis [3], written by Captain Thomas Devenney in 1976 at the Air Force Institute of Technology, examined 16 major acquisition programs at the Electronic Systems Division (ESD) at Hanscom AFB, Massachusetts. He found that in every case the contractor manned up as soon as possible and that the manning level was constant throughout the program. The absence of a demonstration/validation phase and the requirement to hold PDR in the first few months of the contract forced this manning profile. About ten years ago, Roberts [4] suggested a revised manning profile and model schedule for the early design activities leading to PDR and CDR (See Figure 13-7). The intent of this manning profile is to allow more time for design analysis and breadboarding of the difficult parts of the job and to perform some work at a level below the level one is attempting to baseline. This allows for the validation of the top level design and performance allocation before building up the development team. The experience on several ESD programs indicates that with existing practice development specifications are not completed until 10-15 months after contract award. This is a true measure of the time needed for validating top level designs (See Figure 13-8).

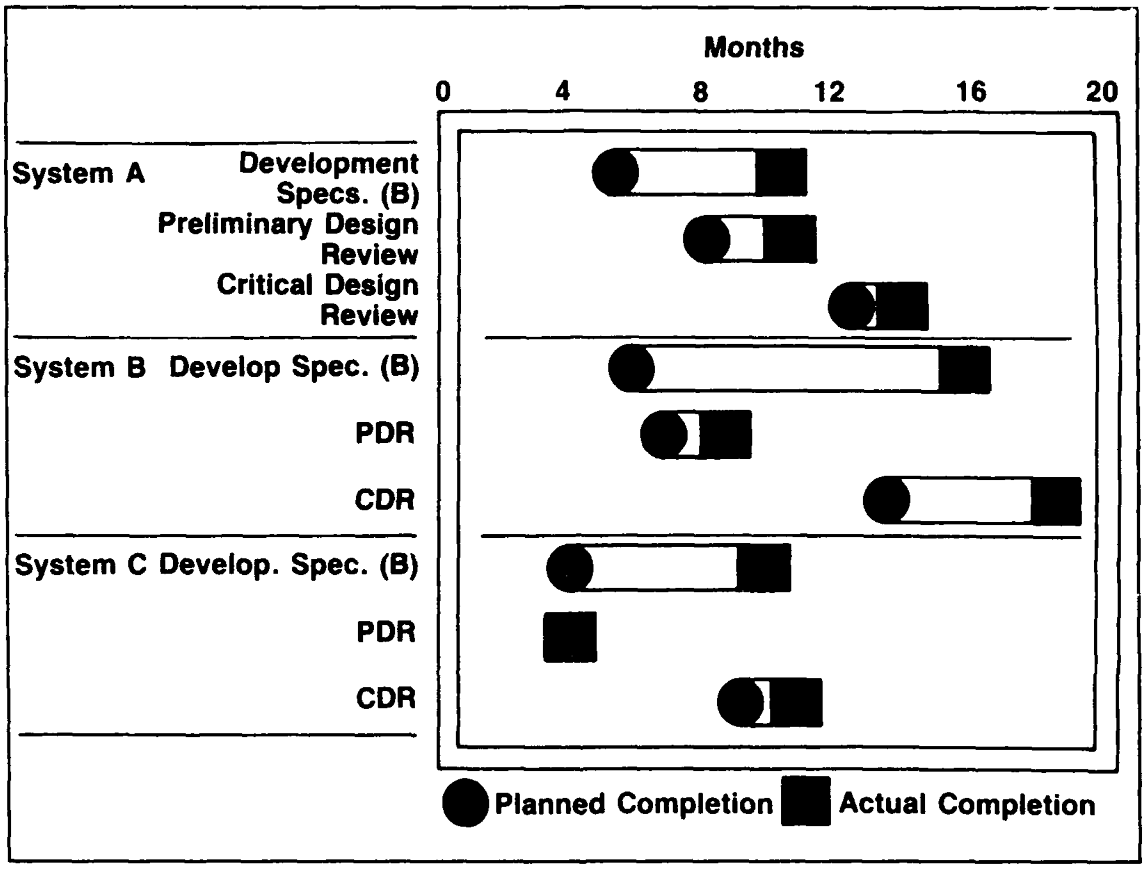


Fig. 13-8 Experience Example

Over time the schedule for the PDR has been increasing and is now generally six months after contract award. What do all those people do while the top level design is being developed? They start into detailed design and coding in order to meet schedule. This premature design and code must often be redone and the changes and uncertainties lead to frustration and poor morale for the entire team.

In updating his COCOMO model for Ada implications, Barry Boehm has suggested a schedule similar to the one shown in Figure 13-7. He allows time before the PDR to compile the PDL statements, validate interfaces, and perform semantic checks. The longer time produces a more orderly process with less breakage and rework.

13.6 TEAM SIZE AND MANAGEMENT

Large, complex software jobs often require hundreds of people. An article [5] by the people who managed Sidewinder, a very successful air-to-air missile, addresses the size and behavior of research and engineering teams. It points out that the nature of complex acquisitions is such that the assignment of too few people may require an infinite time to complete the job. On the other hand, the assignment of additional engineering personnel above a certain level may not only proportionately reduce total time, but may, in fact, increase the total time to accomplishment (See Figure 13-9).

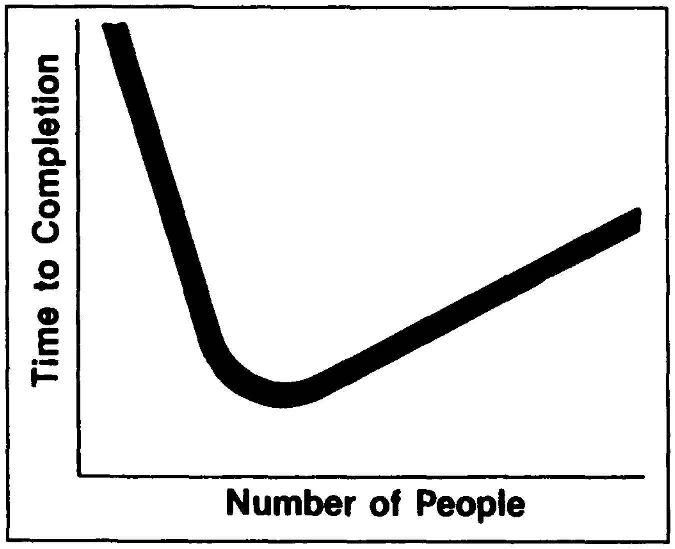


Fig. 13-9 Productivity

The author of the referenced article recalls being asked in grade school to solve the following problem: "If two men can dig a well in eight hours, how long does it take four men to dig the same well". He "... can recall being haunted by the suspicion that perhaps there was only room down the well for two men; in which case the extra two men might have some difficulty in usefully contributing". A similar phenomenon has been reported by Brooks in "The Mythical Man-Month" [6].

If the schedule for a previous job was longer than desired as the result of over-staffing, the tendency may be to add more people to improve the schedule on the next job. This will happen if the inter-dependent nature of the tasks and the over-staffing are not understood. Care must be taken in using past experience to set objectives for future jobs.

The size job a given organization can handle is not only a function of the available number of qualified and experienced personnel. To some extent, jobs can be balanced with resources. When the job grows in size, complexity, and interdependence, the developer must provide appropriate infrastructures (i.e., organization and documentation) to support technical interchange, progress reporting, and increased span of control (See Figure 13-10). If a large job is broken into pieces to get it accomplished, a mechanism must be devised for pulling it back together again.

Failures often occur when firms take on larger jobs than they have previously undertaken. They generally use a small team and face-to-face communications for coordination and control. When the number of people grows, there are no established procedures, tools or documentation to support the necessary technical and management interchanges.

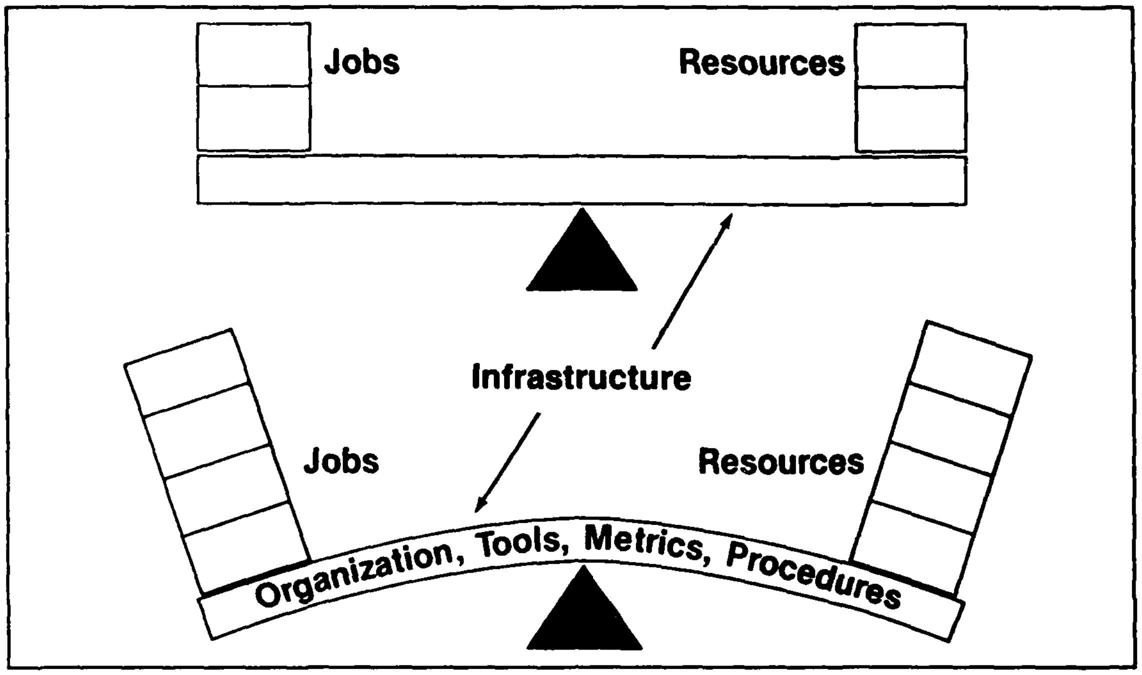


Fig. 13-10 Management

13.7 ASSESSING PERFORMANCE

There are a number of new techniques for assessing a contractor's experience, tools and procedures. Visits to contractor facilities by a team of experienced government software personnel during source selections are being used. The team assesses and verifies contractor experience, maturity of procedures, quality management capabilities and job understanding. At the Electronic Systems Division these teams are called "Greybeards." At the Aeronautical Systems Division, they are called "Capability/Capacity Review Teams" (See Chapter 8). Work at the Software Engineering Institute and at MITRE Corporation [7] has led to a process for assessing the software engineering capability of contractors. A short software engineering exercise has been designed to audit the contractor's use of proposed tools and procedures.

In addition, a set of metrics has been developed for use by the contractor during the software development process as an aid visibility and control [8]. A number of contracts now require the use and reporting of metrics. The cost for this should not be high since a competent contractor normally generates metrics for his own internal use.

13.8 PROGRAM MANAGEMENT GUIDANCE

With present techniques, software development and acquisition is a tough job and will remain so for several years. A number of actions are required to overcome the roadblocks associated with the present process.

<u>Choose a good contractor</u>. During source selection use in-plant inspections (greybeards), software exercises, and contractor capability assessment techniques to aid the selection process. Include the support requirements for these activities in the RFP.

<u>Allow time for design and iteration</u>. Don't baseline without some experience at lower levels of design.

<u>Make maximum use of off-the-shelf software</u>. Try to make the job smaller and more manageable. Change the requirements for well defined, less critical functions to fit with available packages.

<u>Breadboard and prototype the difficult parts of the job</u>. Prototypes help the user see what he is getting and help the developer understand implementation difficulties. Prototypes generally are not suitable for the end product unless they have been especially designed for that purpose.

<u>Schedule preliminary design reviews consistent with the degree of validation</u>. Don't force contractor top level designs to be accomplished in the proposal stage. Allow time to validate the top level design.

<u>Apply discipline and new tools</u>. Contractors should have a good software development plan, established procedures, and tools to aid development, configuration management, test generation, and status keeping.

<u>Require metrics for visibility and control</u>. The contractor should have his own means for determining status and the rate of progress. Trends are the basis for predicting future progress. Impose the requirement to generate and use metrics on the contractor.

<u>Have more than one phase</u>. Have a place to put new requirements as opposed to impacting ongoing efforts. Use the second phase t) deal with shortfalls in the first phase. Often the user no longer wants the same capabilities which were thought desirable early in the process.

<u>Deliver in useful increments</u>. Try to keep the deliveries consistent with the degree of knowledge one has developed of requirements and implementation difficulties. To avoid retraining, design user interfaces so that they don't change with each increment.

<u>Maintain schedule; deliver, then add</u>. Every change in schedule opens the door to new requirements. Delivery provides feedback on performance which is necessary for further evolution.

13.9 REFERENCES

1. Boehm, Barry W., Software Engineering Economics, Prentice Hall, 1981.

2. Roberts, Alan J., "Some Software Implications of System Acquisition, Signal Magazine, July 1982, pages 19-25.

3. Devenney, Capt. Thomas J., "An Exploratory Study of Software Cost Estimating at the Electronic Systems Division," Masters Thesis, Air Force Institute of Technology, July 1976.

4. Roberts, Alan J., "ESD System Acquisition Practices - Design Reviews", The MITRE Corporation, Internal Memo, May 1977.

5. Kirschner. R. B., "The Size of Research and Engineering Teams," Proceedings of the 11th National Conference on the Administration of Research, Pennsylvania State University Press, September 1957.

6. Brooks, F. P., The Mythical Man Month, Addison-Wesley, 1975.

7. A Method for Assessing the Software Engineering Capability of Contractors, Software Engineering Institute, ESD-TR-87-186, CMU/SEI-87-TR-23, 23 September 1987.

8. Schultz, Herman P., Software Management Metrics, ESD-TR-88-011, May 1988.

APPENDIX A
LIST OF ACRONYMS

ADP	Automatic Data Processing
AFSCP	Air Force System Command Pamphlet
AFLC	Air Force Logistics Command
AFB	Air Force Base
AFR	Air Force Regulation
AIS	Automated Information Systems
AMC	Army Materiel Command
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
ASDP	Aeronautical Systems Division Pamphlet
ATE	Automatic Test Equipment
BIT	Built-in-test
BITE	Built-in-test Equipment
CCB	Configuration Control Board
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CE	Concept Exploration
CLIN	Contract Line Item Number
CI	Configuration Item
CIDS	Critical Item Development Spec
CM	Configuration Management
CMP	Configuration Management Plan
CPU	Central Processing Unit
CRISD	Computer Resources Integrated Support Document
CRWG	Computer Resources Working Group
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSOM	Computer System Operator's Manual
CSU	Computer Software Unit
CRLCMP	Computer Resources Life Cycle Management Plan
CRWG	Computer Resources Working Group
DARCOM	U.S.Army Material Development & Readiness Command
DCP	Decision Coordination Paper
DID	Data Item Description
DOD	Department of Defense
DODD	Department of Defense Directive
DOD-STD	Department of Defense Standard
DSB	Defense Science Board
DT&E	Developmental Test and Evaluation
D/V	Demonstration and Validation
ECP	Engineering Change Proposal
ECR	Embedded Computer Resources
EPROM	Erasable Programmable Read Only Memory
EEPROM	Electrically Eraseable PROM
FAR	Federal Acquisition Regulations
FCA	Functional Configuration Audit
FQR	Formal Qualification Review

FQT	Formal Qualification Test
FSD	Full Scale Development
FSM	Firmware Support Manual
HOL	Higher Order Language
HWCI	Hardware Configuration Item
IC	Integrated Circuit
ICBM	Inter-Continental Ballistic Missile
ICWG	Interface Control Working Group
IDD	Interface Design Document
IEEE	Institute of Electric and Electronic Engineering
ILSP	Integrated Logistics Support Plan
I/O	Input/Output
IOC	Initial Operating Capability
IR&D	Independent Research and Development
IRS	Interface Requirement Specification
ISA	Instruction Set Architecture
IV&V	Independent Verification and Validation
JLC	Joint Logistics Commanders
MCCR	Mission Critical Computer Resources
MIL-STD	Military Standard
NBS	National Bureau of Standards
OFP	Operational Flight Program
OMB	Office of Management and Budget
OSD	Office of Secretary of Defense
OT&E	Operational Test and Evaluation
PC	Personal Computer
PCA	Physical Configuration Audit
PDL	Program Design Language
PDR	Preliminary Design Review
PDSS	Post-Deployment Software Support
PIDS	Prime Item Development Spec
PM	Program Manager
PMD	Program Management Directive
PMP	Program Management Plan
PMRT	Program Management Responsibility Transfer
PO	Program Office
PROM	Programmable Read Only Memory
QA	Quality Assurance
RAM	Random Access Memory
R&D	Research and Development
RF	Radio Frequency
RFP	Request for Proposal
ROM	Read Only Memory
SCM	System Concept Paper
SCP	System Concept Paper
SCRB	Software Configuration Review Board

SDCCR	Software Development Capability/Capacity Review
SDD	Software Design Document
SDF	Software Development Folder (File)
SDL	Software Development Library
SDP	Software Development Plan
SDR	System Design Review
SEI	Software Engineering Institute
SIL	Systems Integration Laboratory
SIF	Systems Integration Facility
SON	Statement of Need
SOW	Statement of Work
SPD	System Program Director
SPM	Software Programmer's Manual
SPO	System Program Office
SPR	Software Problem Report
SPS	Software Product Specification
SRR	Systems Requirements Review
SRS	Software Requirements Specification
SSA	Software Support Activity
SSA	Source Selection Authority
SSAC	Source Selection Advisory Council
SSEB	Source Selection Evaluation Board
SSDD	System/Segment Design Document
SSP	Source Selection Plan
SSR	Software Specification Review
SSS	System/Segment Specification
STARS	Software Technology for Adaptable Reliable Systems
STD	Standard
STE	Special Test Equipment
STP	Software Test Plan
STR	Software Test Report
SUM	Software User's Manual
S/W	Software
TADSTAND	Tactical Digital Systems Standard
T&E	Test and Evaluation
TEMP	Test and Evaluation Master Plan
TO	Technical Order
TPS	Test Program Set
TRR	Test Readiness Review
UDF	Unit Development Folder
VDD	Version Description Document
VHSIC	Very High Speed Integrated Circuit
WBS	Work Breakdown Structure

APPENDIX B
GLOSSARY OF TERMS

Address
Specifies the location of word, data or instruction in memory.

Allocated Baseline
The development specification which defines performance requirements for each CSCI.

Analog
Being or relating to a mechanism in which data is represented by continuously variable physical quantities [1].

Assembler
A computer program that translates assembly language instructions into machine language. When an assembler is used, typically one assembly language instruction is translated into one corresponding machine language instruction. Both the assembly and machine languages are unique to a particular computer.

Assembly Language
Assembly language allow the use of abbreviated names (mnemonics) for machine language instructions and operands in place of binary (0s and 1s) machine codes.

Bit
A binary digit whose value is either a 1 or a 0.

Built-in Test Equipment (BITE)
Any device permanently mounted in the prime equipment and used for the express purpose of testing the prime equipment, either independently or in association with external test equipment.

Byte
An eight (8) bit word.

Buss
See Digital Data Buss

Central Processing Unit (CPU)
Fetches, decodes and executes the instructions of the computer program.

Compilation or Compiling
The translation process accomplished by a compiler.

Compiler

A computer program which translates a HOL into machine language. The HOL statements are called source code and the output of the compiler is called object code.

Component

A Computer Software Component (CSC) is a distinct part of a computer software configuration item (CSCI). CSCs may be further decomposed into other CSCs and CSUs.

Computer Program

A series of instructions or statements in a form acceptable to computer equipment and designed to cause the execution of an operation or series of operations. Computer programs include such items as operating systems, assemblers, compilers, interpreters, data management systems, utility programs, and maintenance or diagnostic programs. They also include application programs such as payroll, inventory control, operational flight, strategic, tactical, automatic test, crew simulator, and engineering analysis programs. Computer programs may be either machine-dependent or machine-independent, and may be general purpose in nature or designed to satisfy the requirements of a specialized process or particular users.

Debugging

The process of locating and eliminating errors that have been shown to exist in a computer program.

Digital Data Buss

A group of circuits and interconnections between two or more devices, such as between the CPU and memory or between the computer and external devices, that provide a communication path for digital data.

Error Message

A message printed out by a computer after detecting a programming error.

Emulator

A combination of computer programs and computer hardware that mimic the instructions and execution speed of another computer.

Executive

The operating system in an avionics suite.

Expert Systems

Systems that utilize artificial intelligence (AI) to perform their functions.

Firmware
Computer programs and data that have been written into read only memories (ROMs).

Formal Qualification Review (FQR)
A system level configuration audit conducted after system testing is completed to ensure that the performance requirements of the system specification have been met.

Functional Baseline
The system requirements. Provides basis for contracting and controlling the system design.

Functional Configuration Audit (FCA)
The formal examination of test data to determine the functional characteristics of a CSCI, prior to acceptance, to verify that the item has achieved the performance specified in its functional or allocated configuration identification.

Higher Order Language (HOL)
Higher order languages have been developed in order to make writing and understanding programs easier. In a HOL, the program is written in a series of statements which typically resemble mathematical formulas or English expressions.

Host Computer
The computer on which a compiler executes.

Integrated Circuit (IC)
Tiny complex of electronic components and their connections that is produced in or on a small slice of material such as silicon [1]. The basic building blocks of modern electronics.

Intermediate Language
An assembly-like language used by a compiler as an interim step in the process of compilation.

Interpreter
A computer program that converts and executes a HOL source program statement directly into machine language, one statement at a time.

Linker
A computer program that links or ties together programs that have been separately compiled or assembled.

Loader
The computer program that loads the computer program into memory.

Machine Language
The binary codes (0s and 1s) which are understood directly by a computer. A typical machine language instruction consists of an operation code (or op-code) and one or more operand fields. The operation code specifies the computer function (e.g., add, subtract, test for zero) to be performed while the operand fields specify where in the computer the data for that function is located.

Microprocessor
A Central Processing Unit (CPU) constructed from one large scale integration device or chip.

Module
See Unit

Module Testing
The execution of a single module to determine its correctness before the module is combined or integrated with other modules.

Mnemonic
Symbolic names for machine language instructions which allow a programmer to generate programs in assembly language without having to use binary codes.

Operating System
A computer program that controls the execution of other computer programs in a computer. It schedules the time when computer programs are run, assigns memory, and provides diagnostic and accounting information about a program's execution.

Patching
Making changes to the machine code (object code) representation of a computer program.

Regression Testing
The testing of the program that is performed to confirm that functions, that were previously performed correctly, continue to perform correctly after a change has been made.

Rehosting
Modifying a computer program so that it operates on a different host computer.

Retargeting
Modifying a compiler so that it generates object code for a different target computer.

Software
The combination of computer programs or instructions required to cause the computer hardware to perform a certain task or tasks.

Stub
A stub takes the place of a module that has not yet been coded or tested.

Syntax
The rules for writing computer programs in a particular programming language.

Syntax Error
A syntax error is generated when a programmer has violated the rules of a particular programming language.

Target Computer
The computer for which the compiler generates object code.

Test Program Set (TPS)
Computer programs written in a HOL, usually ATLAS, used in conjunction with automatic test equipment (ATE) to isolate a failed electronic subsystem or component. TPSs typically are used to generate and inject test patterns into digital circuit boards.

Unit
A Computer Software Unit (CSU) is the smallest element specified in a Computer Software Component (CSC) that is separately testable.

Validation
The process of confirming that the software (i.e., documentation and computer program) satisfies all user requirements when operating in the user's environment.

Verification
The process of confirming that the products of each software development phase (e.g., requirements analysis, design, coding) are complete, correct, and consistent with respect to the products of the previous phase.

Word
A data packet of information for the computer; it is usually composed of many bits. The length of a computer word typically ranges from 8 bits for microprocessors to 64 or more bits for the larger computers. The memory of a computer is divided into segments called words.

REFERENCES

1. Webster's Ninth New Collegiate Dictionary, Merriam-Webster Inc., Springfield, MA, 1984.

APPENDIX C
OUTLINE OF PROGRAM MANAGEMENT PLAN

Table C-1 shows a typical outline of a PMP based on AFR 800-2, "Acquisition Program Management", Attachment 3. The other services use a similar outline. Note that six of the sections contain an asterisk to indicate that these sections must address mission critical computer resources (MCCR). This doesn't mean that these are the only sections that address MCCR. Depending on the program, there may be other sections that may need to address software.

SECT		SUBJECT
1		Program Summary and Authorization
2	*	Intelligence
3	*	Program Management
4	*	Systems Engineering & Configuration
5	*	Test and Evaluation
6		Information Systems
7		Operations
8		Civil Engineering
9		Logistics
10	*	Manpower and Organization
11		Personnel Training
12		Security
13	*	Directives, Specifications and Standards

* Addresses MCCR

Table C-1 PMP Outline

C.1 INTELLIGENCE (Sect 2)

This section includes:

(a) Identification of the Threat. This paragraph should consist of a listing of all relevant threats that have been obtained from the Defense Intelligence Agency (DIA) or other DOD or armed service agencies. Since software will probably be the major player in countering these threats, it is important that the entire spectrum of threats be addressed.

(b) Identification of Relevant Foreign Technology. Since the U.S. is no longer the undisputed world leader in technology, it is important that foreign technology, especially that of our allies, be examined to determined whether any of it can be used in the proposed system. Our European allies, for example, have embraced Ada, the DoD standard computer language, with fervor and have made significant strides in the area of Ada software tools, some of which could be useful.

C.2 PROGRAM MANAGEMENT (Sect 3)

This section provides a description of the objectives and program strategy (or approach) in somewhat more detail than Section 1. At a minimum, the schedules contained in this section should include the following:

(a) Operational system software development schedules

(b) Training schedules including the procurement and development of all maintenance and operational crew trainers

(c) Support equipment development and delivery schedule including ATE and other software intensive special test equipment

(d) Development and delivery schedule for the TPSs associated with the ATE.

C.3 SYSTEMS ENGINEERING & CONFIGURATION MANAGEMENT (Sect 4)

This section should describe when and how the functions of hardware and software configuration management will be accomplished to include the Configuration Control Board (CCB) and the Software CCB.

C.4 TEST AND EVALUATION (Sect 5)

This section addresses DT&E and OT&E schedules including the overall software test and integration schedules. It also describes the major software support tools and major test facilities required.

C.5 MANPOWER AND ORGANIZATION (Sect 10)

This section describes the organization of the program office and summarize the relationships and roles of other military and government agencies and laboratories. In particular, the software organization and its relation to the other program office organizations should be described. The required software manpower and software skill levels should also be addressed along with possible sources of key software personnel.

C.6 DIRECTIVES, SPECIFICATIONS, AND STANDARDS (Sect 13)

This section lists all the directives, specifications, and standards that will be imposed on the program including those related to software and computers. For example, if the data buss standard, MIL-STD-1553B, or the Ada programming language standard, MIL-STD-1815, is inappropriate, the reasons why should be stated in this section and the waiver process initiated.

APPENDIX D
TEST AND EVALUATION MASTER PLAN OUTLINE

The formal outline for the TEMP is given in DOD Directive 5000.3-M-1 and it is shown in Table D-1. The TEMP is broken up into five major sections and four appendices. Software must be addressed in sections III.1.e , V.7, and Appendix 3.

PART		SUBJECT
I		System Details
II		Program Summary
III		DT&E Outline
III.1.e	*	Software Test & Evaluation
IV		OT&E Outline
V		T&E Resource Summary
V.7	*	Simulators, Models & Testbeds (MCCR)
APPENDICES		
1		References
2		Bibliography
3	*	Software T&E Plan (Embedded Computers)
4		Definitions

* Addresses MCCR

Table D-1. TEMP Outline

D.1 SOFTWARE TEST & EVALUATION (Part III.1.e)

This section describes the software testing of all the mission critical computer resources required to demonstrate a quality product, including post-milestone III updates. When identifying the MCCR, only non-embedded computer resources are addressed. Embedded resources are addressed in Appendix 3. Some of the non-embedded MCCR resources to be identified include:

(a) All data reduction and analysis computers and software required for T&E;

(b) All special test equipment (STE) containing computers and software;

(c) All the automatic test equipment (ATE) and associated software required to test and maintain all of the weapon system electronics equipment;

(d) All the system and subsystem trainers required to train the weapon system operational and maintenance crews.

D.2 TEST & EVALUATION RESOURCE SUMMARY (Part V)

This section provides a summary of all key resources, both government and contractor, required during the course of the acquisition program. The initial TEMP should project those key resources, including major range and unique instrumentation requirements, threat simulators, and targets, necessary to accomplish DT&E and OT&E objectives. As system development progresses, test resource requirements should be reassessed and subsequent TEMP updates should reflect any changed system concepts or requirements, and/or updated threat assessments. Specifically, **Part V.7, Simulators, Models, and Testbeds**, identifies, for each specific test and evaluation phase, the system simulations required, including computer driven simulation models and hardware-in-the-loop testbeds, to conduct the planned test program. These system simulation requirements should be compared with existing and programmed capabilities and any major shortfalls should be highlighted.

D.3 SOFTWARE T&E PLAN (Appendix 3)

This section describes the software testing necessary to demonstrate the ability of embedded MCCR to achieve system objectives. As the program development progresses, this initial Software T&E Plan is incorporated intc the software developer's test plans and the Software Development Plan.

APPENDIX E
INTEGRATED LOGISTICS SUPPORT PLAN OUTLINE

Table E-1 shows a typical outline of an ILSP. Guidance for this document is found in AR 700-127, Integrated Logistics Support, AFR 800-8, Integrated Logistics Support Program, and SECNAVINST 5000.39, Integrated Logistics Support in the Acquisition Process. The sections marked by an asterisk sections may address MCCR.

PART		SUBJECT
I		General
(1)		System Description
(2)		Program Management
(3)		Applicable Documents
II		Concepts/Strategy
(1)		Operational and Organizational Concept
(2)	*	Maintenance Concept
(3)		Logistics Support Analysis
(4)	*	Acquisition Strategy
(5)		Test and Evaluation
(6)		Spares Acquisition Concept
(7)		Planning for Deployment
(8)		Support Resource Funds
(9)		Post Fielding Assessments
(10)	*	Computer Resources
(11)	*	Training
III		Milestones Schedule Charts

* Addresses MCCR

Table E-1 ILSP Outline

E.1 MAINTENANCE CONCEPT (Part II.2)

This section normally describes the three traditional levels of maintenance: organizational or field level maintenance, intermediate level maintenance, and depot level maintenance. Since modern weapon systems are heavily dependent on electronics, this section will discuss the types and number of automatic test equipment (ATE) contemplated for the three levels of support; the associated Test Program Sets (TPS) which are used in conjunction with the ATE to isolate failed electronics components; and any other computer and software dependent piece of calibration or test equipment.

E.2 ACQUISITION STRATEGY (Part II.4)

This section merely summarizes the acquisition strategy described in the Acquisition Plan. For software and computers, this section should briefly describe how the software for the logistics support will be acquired. For example, who will develop the ATE and TPSs, the prime developer, a subcontractor, or another government agency? Will the program office delegate the responsibility for acquiring the ATE and TPSs to another government agency (a common occurrence in the Air Force)? Bear in mind that development by anyone other than the prime contractor involves additional contractual considerations (e.g., separate RFPs, source selections, and contracts). How will calibration and special logistics support test equipment be acquired?

E.3 COMPUTER RESOURCES (Part II.10)

This section describes the facilities, hardware, software, documentation, manpower, and personnel needed to operate and support the embedded computer systems. Particular attention should be paid to the ATE and the associated TPSs required to isolate problems in Line Replaceable Units (LRUs) and electronic circuit cards. Organic maintenance and support cannot be initiated until the ATE and the TPSs are delivered.

E.4 TRAINING (Part II.11)

This section addresses the requirements for training all the personnel required to support the system. Since this usually requires complex and computer intensive trainers, the various types and numbers of trainers, their anticipated location, and availability dates should given. Examples of some of the potential trainers required are: avionics maintenance trainers, weapons loading trainers, and ATE trainers.

APPENDIX F
COMPUTER RESOURCES LIFE CYCLE MANAGEMENT PLAN OUTLINE

Table F-1 shows the CRLCMP format found in Attachment 11 of AFR 800-14, "Life Cycle Management of Computer Resources in Systems". Although the other services do not have a standard format, the formats used are very similar.

SECT	SUBJECT
1.	Introduction
a.	Overview
b.	Scope and Applicability
c.	References
2.	Systems Concept
a.	Operational Concept
b.	Support Concepts
3.	System Description
a.	Overview
b.	Computer Hardware
c.	Computer Software
4.	Computer Resources Design
a.	System Architecture and Integration
b.	Product Improvements
c.	Software Development Tools
d.	Reusability
e.	Interoperability
f.	Additional Design Constraints
5.	Organizational Roles
a.	Implementing Command
b.	Supporting Command
c.	Operating Command
d.	Using Command (If Applicable)
e.	Participating Commands
f.	Other Agencies
6.	Resources
a.	Personnel
b.	Facilities
c.	Training
d.	Hardware
e.	Software
f.	Integrated Logistics Support
7.	Documentation
a.	Types of Documents
b.	Data Rights
c.	Data Management
8.	Acquisition Management Practices
a.	Software Development Strategy
b.	Boards and Committees
c.	Configuration Management
d.	Documentation Review or Approval
e.	Reviews and Audits
f.	Test and Evaluation
g.	Software Quality
h.	Security
9.	Transition Management Practices
a.	Configuration Management
b.	Turnover
c.	Support During Transition
d.	Transfer
10.	Deployment Management Practices
a.	Boards and Committees
b.	Configuration Management
c.	Security
d.	Training
11.	Schedules
a.	Major Milestones
b.	Contract Delivery Schedule
c.	Support Capabilities
Appendices	
A	Acronyms and Abbreviations
B	Glossary of Terms
C	List of Key Personnel
D	CRWG Charter
E	Risk Management Plan
F	Detailed System Description
G	Security Assistance

Table F-1 CRLCMP Outline

F.1 INTRODUCTION

This section states the purpose of the CRLCMP, lists the approved system nomenclature, and lists the appropriate requirements documents such as the Statement of Need (SON) and the System Operational Concept (SOC).

F.2 SYSTEM CONCEPTS

This section describes the system operational and support concepts. It briefly describes the mission of the system with emphasis on computer resources; identifies the system functions which are expected to require frequent changes to accommodate the operational environment; and describes the hardware support concept for the system and for the software.

F.3 SYSTEM DESCRIPTION

States the purpose of the operational system and describes how the computer resources relate to the overall operational system. Identifies and describes the characteristics and functions of the processors in the system and the functions to be implemented in software or firmware.

F.4 COMPUTER RESOURCES DESIGN

This section addresses the following:

(a) System Architecture and Design - Identifies the required hardware and software architectures for the system.

(b) Product Improvements - Identifies parts of the system which will most likely require future expansion (e.g., memory size, processing capacity, number of interfaces).

(c) Software Development Tools - Identifies and briefly describes the required software development tools and their environment for usage. Indicates whether each tool is GFE, or GFP, commercially available or contractor developed.

(d) Reusability - Identifies and briefly describes any developed software, tools, or facilities that may apply to other current or future systems.

(e) Interoperability - Briefly describes any interoperability requirements for the system that are implemented in computer resources.

(f) Additional Design Constraints - Briefly describes additional performance and support constraints and considerations which must be translated into specified requirements.

F.5 ORGANIZATION ROLES

This section describes the functional relationships among the implementing command, the supporting command, the operating command, and any other participating command or government agency.

F.6 RESOURCES

This section identifies the personnel, facilities, training, hardware, software, and integrated logistics support requirements for the system.

F.7 DOCUMENTATION

This section summarizes the documentation requirements, identifies the government's software data rights, and describes the plans and procedures for managing the data.

F.8 ACQUISITION MANAGEMENT PRACTICES

This section discusses the software development strategy; identifies configuration control boards and their interfaces to other boards or committees; identifies the existing directives which will govern configuration management activities; identifies the schedule and participating organizations for computer resource related reviews and audits; and describes the schedule and participating organizations for test and evaluation of computer resources.

F.9 TRANSITION MANAGEMENT PRACTICES

This section discusses the procedures and directives which will govern configuration management and support during and after the system is transferred from the developing organization to the using and supporting organizations.

F.10 DEPLOYMENT MANAGEMENT PRACTICES

This section identifies the boards and committees which are created for the management of computer resources during the acquisition phase; identifies the existing directives governing security for the system; and describes the activities and major milestones associated with training personnel for operating and supporting the system.

F.11 SCHEDULES

This section identities the major milestones associated with the acquisition, transition, and support schedules of the system; identifies the schedules associated with contract deliverables; and identifies the schedule for the initial operating capability for the primary computer resource support capabilities.

F.12 APPENDICES

The appendices will contain the complete charter for the Computer Resources Working Group (CRWG) as well as the other items indicated in the CRLCMP outline of Table F-1. It is extremely impoitant that the program manager carefully review the CRWG charter prior to endorsing it. The CRWG must reflect the program manager's objectives and philosophy. Failure of the program manager to scrutinize this charter can lead to misunderstandings and/or wasted efforts later in the program.

APPENDIX G
SOURCE SELECTION PLAN

Table G-1 shows an outline of a Source Selection Plan (SSP) as given in AFR 70-15, "Source Selection Policy and Procedures". The other services use a similar format. The asterisk indicates those sections that should address mission critical computer resources.

SECT		SECT
1		Introduction
2	*	Source Selection Organization
3	*	Screening Criteria
4		Evaluation Process
5	*	Evaluation Criteria
6	*	Acquisition Strategy
7		Schedule of Events

* Addresses MCCR

Table G-1 SSP Outline

G.1 SOURCE SELECTION ORGANIZATION (Sect 2)

This section describes the Source Selection Authority (SSA), the Source Selection Advisory Counsel (SSAC), and the Source Selection Evaluation Board (SSEB organizations and lists recommended key members by name, or by position or functional area. It must also specify other government organizations that will be represented on the SSAC and SSEB, and include an estimate of the total number of personnel who will form the membership, including any advisors.

It is important that the key software person on the SSEB be very knowledgeable about software acquisition. Finding this individual will not be easy since knowledgeable, senior software individuals are not always readily available. If this individual is a member of the program office, then his or her availability will not be a problem. However, if no such individual can be found within the program office, searching for that individual must become a matter of high priority.

If the program office is newly formed and without a knowledgeable software individual, the program manager must look to either other program offices, laboratories, or other external organizations for an individual who can spend one to three months (a typical duration period for a source selection) away from his or her current job. The longer the source selection, the more difficult it will be to find this individual. Ideally the senior software person involved in the source selection should be the same individual who will be involved in managing the effort once a developer is selected. Since the selection of the right software developer is paramount to the success of the entire program, finding a knowledgeable software evaluator should be the number one priority of the program manager.

G.2 SCREENING CRITERIA (Sect 3)

This section describes the criteria to be used to select prospective sources. The screening criteria must be developed prior to the official publication of the planned procurement effort in the Commerce Business Daily (CBD). It will be used by procurement personnel to determine potential sources based on the letters of interest received in response to the CBD announcement. Contractors who have expressed interest and satisfy the screening criteria will receive copies of the proposal package. Even if a contractor has failed to satisfied the screening criteria, it may still receive a copy of the proposal if it requests it. Ordinarily, however, not too many contractors wish to pursue a procurement in which the government doesn't feel it is competitive.

The screening criteria must include the requirement that the sources solicited will have (inherently or by subcontracting or teaming arrangements) the management, financial, technical expertise, and necessary security clearances to design and develop the overall system software.

G.3 EVALUATION CRITERIA (Sect 5)

This section describes the specific evaluation criteria which will be used to judge all of the proposals. In generating the evaluation criteria, the program office must ensure that criteria is not overly restrictive or biased towards a particular approach or technology. Remember that the developer should have the latitude to select both the technology and approach that best satisfies the stated requirements. Unless there are valid reasons for forcing the contractor to pursue a particular path, such as compatibility with an existing system, the evaluation criteria should reflect the general requirements and not a particular design.

G.4 ACQUISITION STRATEGY (Sect 6)

This section will include a summary of the acquisition strategy, including type of contract(s) proposed, the incentives contemplated, milestones demonstrations intended, and special contract clauses to be used.

A strategy that may be used is to make the prime contractor responsible for developing all of the system software or subcontracting it out. Another approach may be to direct the prime contractor to subcontract critical portions of the software to specialized software houses. For example, a defensive avionics subsystem may be subcontracted out to a subcontractor specializing in electronic warfare systems.

With the increasing role played by digital electronics subsystems, an important segment of software is the development of Test Program Sets (TPS) for automatic test equipment (ATE). Since TPS development is a highly specialized business, a prime contractor normally does not have the expertise to develop them. Even if he did, it is not prudent to put so many software eggs in one basket. It may be highly advisable to contract separately for TPS development. This will mean a separate source selection effort.

Other specialized software areas are trainers and simulators. Once again it may be prudent to contract these efforts separately. For a major simulator such as a weapon systems trainer, the program office may delegate this responsibility to another organization or program office.

Whatever strategy is selected, the most likely approach will involve more than one contractor. It is important that the proposal clearly state the government's preference so that contractors include in their proposals the efforts required to coordinate with other contractors and to provide them with technical data.

Finally this section should state whether the program office will use an Independent Verification and Validation (IV&V) organization and what their level of involvement with the software developers.

APPENDIX H
SOFTWARE DATA ITEM DESCRIPTIONS (DIDs)

DID	TITLE
Specifications:	
DI-CMAN-80008A	System/Segment Specification (SSS)
DI-CMAN-80534	System Segment Design Document(SSDD)
DI-MCCR-80025A	Software Requirements Specification (SRS)
DI-MCCR-80026A	Interface Requirements Specification (IRS)
DI-MCCR-80029	Software Product Specification (SPS)
Documentation:	
DI-MCCR-80030A	Software Development Plan (SDP)
DI-MCCR-80012a	Software Design Document (SDD)
DI-MCCR-80013A	Version Description Document (VDD)
DI-MCCR-80014A	Software Test Plan (STP)
DI-MCCR-80015A	Software Test Description (STD)
DI-MCCR-80017A	Software Test Report (STR)
DI-MCCR-80018A	Computer System Operator's Manual (CSOM)
DI-MCCR-80019A	Software User's Manual (SUM)
DI-MCCR-80021A	Software Programmer's Manual (SPM)
DI-MCCR-80022A	Firmware Support Manual (FSM)
DI-MCCR-80024A	Computer Resources Integrated Support Document (CRISD)
DI-MCCR-80027A	Interface Design Document (IDD)
DI-MCCR-80029A	Software Product Specification (SPS)

U.S.GOVERNMENT PRINTING OFFICE:1988-201-454/92397

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION Defense Systems Management College	6b. OFFICE SYMBOL *(If applicable)* SE-T	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State, and ZIP Code)* DSMC/SE-T Fort Belvoir, VA 22060-5426		7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Defense Systems Management College	8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)* DSMC-SE-T Fort Belvoir, VA ^2060-5426	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*

Mission Critical Computer REsources Management Guide (U)

12. PERSONAL AUTHOR(S)
I. Caro, R. Higuera, F. Kockler, S. Jacobson, A. Roberts

13a. TYPE OF REPORT	13b. TIME COVERED FROM ________ TO ______	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT 224

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*	
FIELD	GROUP	SUB-GROUP	Computers	Metrics
			Software	Software Support
			Weapon Systems	S/W Development Cycle

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

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] [✓ SAME AS RPT.] ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Israel I. Caro	22b. TELEPHONE *(Include Area Code)* (803) 664-5198	22c. OFFICE SYMBOL SE-T

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

Comment Sheet
for
Mission Critical Computer Resources Management Guide

This guide provides a working familiarity with mission critical computer resources, their development and acquisition. The software development process is discussed in detail along with management principles for timely and cost-effective software acquisition. Your comments and suggestions are solicited.

If you have comments, please tear this sheet out, write the comments in the space provided below, fold, tape closed, and mail. This form is pre-addressed and needs no postage.

Comment:

Name/Title____

Address ____

____

Telephone (Commercial)______________**(AUTOVON)**______________**(FTS)**__________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.12062 FORT BELVOIR, VA

POSTAGE WILL BE PAID BY ADDRESSEE

Defense Systems Management College
Attn: DRI-P
Fort Belvoir, VA 22060-9989