

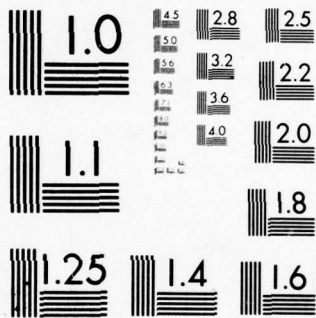
AD-A053 040

SYSTEM DEVELOPMENT CORP SANTA MONICA CALIF  
SOFTWARE ACQUISITION MANAGEMENT GUIDEBOOK: SOFTWARE MAINTENANCE--ETC(U)  
OCT 77 J R STANFIELD, A M SKRUKRUD F19628-76-C-0236  
SDC-TM-5772/004/02 ESD-TR-77-327 NL

UNCLASSIFIED

1 OF 1  
AD  
A063040





MICROCOPY RESOLUTION TEST CHART  
 NATIONAL BUREAU OF STANDARDS-1963-A

2



AD A 053040

SOFTWARE ACQUISITION MANAGEMENT  
GUIDEBOOK: SOFTWARE MAINTENANCE

J. R. Stanfield  
A. M. Skrukud

October 1977

AD No. \_\_\_\_\_  
DDC FILE COPY

Approved for Public Release;  
Distribution Unlimited.

DDC  
APR 24 1978  
RECEIVED  
F

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS  
ELECTRONIC SYSTEMS DIVISION  
HANSCOM AIR FORCE BASE, MA 01731

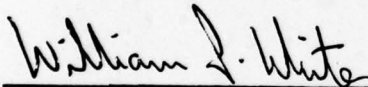
LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

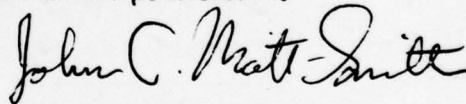
OTHER NOTICES

Do not return this copy. Retain or destroy.

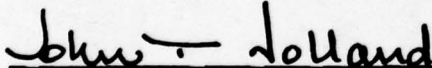
This Technical Report has been reviewed and is approved for publication.



WILLIAM J. WHITE, Capt, USAF  
Project Engineer

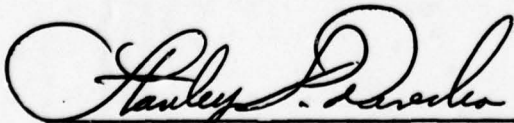


JOHN C. MOTT-SMITH  
Project Manager



JOHN T. HOLLAND, Lt Col, USAF  
Chief, Techniques Engineering Division

FOR THE COMMANDER



STANLEY P. DERESKA, Colonel, USAF  
Deputy Director, Computer Systems  
Engineering

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

<b>19</b> REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
<b>18</b> 1. REPORT NUMBER ESD-TR-77-327	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER <i>kept.</i>	
<b>6</b> 4. TITLE (and Subtitle) SOFTWARE ACQUISITION MANAGEMENT GUIDEBOOK: SOFTWARE MAINTENANCE	<b>9</b> 5. TYPE OF REPORT & PERIOD COVERED Contract June 1976 to Oct 1977	6. PERFORMING ORG. REPORT NUMBER <b>14</b> SDC-TM-5772/004/82	
<b>10</b> 7. AUTHOR(s) J. R. Stanfield A. M. Skrukud	8. CONTRACT OR GRANT NUMBER(s) <b>15</b> F19628-76-C-0236		<b>16</b>
9. PERFORMING ORGANIZATION NAME AND ADDRESS SYSTEM DEVELOPMENT CORPORATION 2500 COLORADO AVENUE SANTA MONICA, CAL. 90406		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE64740F, Project 2238	
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command & Management Systems Electronic Systems Division Hanscom AFB, Mass. 01731		<b>11</b> 12. REPORT DATE OCT 1977	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) <b>12</b> 66p.		13. NUMBER OF PAGES 62	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; Distribution Unlimited		15. SECURITY CLASS. (of this report) UNCLASSIFIED	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
18. SUPPLEMENTARY NOTES			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) COMPUTER PROGRAM MAINTAINABILITY      SOFTWARE MAINTAINABILITY COMPUTER PROGRAM MAINTENANCE          SOFTWARE MAINTENANCE MAINTAINABLE SOFTWARE MAINTENANCE			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report is one of a series of Software Acquisition Management (SAM) guidebooks which provide information and guidance for ESD Program Office personnel who are charged with planning and managing the acquisition of command, control, and communications system software procured under Air Force 800 series regulations and related software acquisition management concepts.			

DDC  
APR 24 1978  
F

339900

JB

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

20. ABSTRACT (cont'd)

The scope of this document is limited to those acquisition and development activities, occurring throughout the SAM cycle, which impact software maintenance. It includes discussions of system turnover to the using command and the transfer of program management responsibility to the supporting command. The computer program life cycle is also considered. Most of the information provided in this report covers the implementing command's responsibilities during the SAM cycle. However, software maintenance during the Deployment Phase is also discussed to provide the background for proper planning. Current programming concepts are discussed as well as the military regulations, specifications, and standards. Within these constraints, this report emphasizes what the Program Office can do to specify and procure maintainable software, including procurement of the facilities, support tools, and documentation necessary to support software maintenance activities.

ACCESSION for	White Section <input checked="" type="checkbox"/>
NTIS	Buff Section <input type="checkbox"/>
DDC	<input type="checkbox"/>
UNANNOUNCED	
JUSTIFICATION	
BY	DISTRIBUTION/AVAILABILITY CODES
	SP CIAL

**A**

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

## PREFACE

The Software Maintenance guidebook is one of a series of Software Acquisition Management (SAM) guidebooks designed to help ESD Program Office personnel in the acquisition of embedded software for command, control and communications systems. The contents of the guidebook will be revised periodically to reflect changes in software acquisition policies and practices as well as feedback from guidebook users.

This report was prepared by System Development Corporation (SDC) under the direction of the Computer Systems Engineering Directorate (MCI) of the Electronic Systems Division (ESD), Air Force Systems Command (AFSC). Contributions were made by: Mr. J. Mott-Smith and Captain W. White (ESD/MCI); Mr. J. Trachtenberg (AFALD/AQE); Mr. M. Landes (RADC/ISI); Mr. M. Mleziva (ESD/EN); Mr. M. Zymaris (ESD/DRT); Mr. D. Peterson (The MITRE Corporation); Captain J. Haughney (AFCS/LO); and Mr. G. Gehlauf (AFLC/LOAK).

The SAM Guidebook series covers the following topics (National Technical Information Service accession numbers for those already published are shown in parentheses):

- Regulations, Specifications and Standards (AD-A016401)
- Contracting for Software Acquisition (AD-A020444)
- Monitoring and Reporting Software Development Status (AD-A016488)
- Statement of Work Preparation (AD-A035924)
- Reviews and Audits
- Computer Program Configuration Management
- Computer Program Development Specification  
(Requirements Specification)
- Software Documentation Requirements (AD-A027051)
- Verification
- Validation and Certification
- Overview of the SAM Guidebooks
- Software Maintenance
- Software Quality Assurance
- Software Cost Estimation and Measurement
- Software Development and Maintenance Facilities (AD-A038234)
- Life Cycle Events (AD-A037115)

## TABLE OF CONTENTS

	<u>Page</u>
PREFACE . . . . .	1
LIST OF FIGURES . . . . .	4
SECTION 1. INTRODUCTION . . . . .	5
1.1 Purpose . . . . .	5
1.2 Scope . . . . .	5
1.3 Software Maintainability . . . . .	7
1.3.1 Definition . . . . .	7
1.3.2 Factors which Support Software Maintainability . . . . .	8
1.4 Contents . . . . .	10
SECTION 2. ACQUIRING MAINTAINABLE SOFTWARE . . . . .	11
2.1 Defining and Specifying . . . . .	11
2.1.1 Planning Considerations . . . . .	11
2.1.2 Development Specifications and the Full-Scale Development Phase RFP . . . . .	14
2.2 Monitoring and Evaluating . . . . .	19
2.2.1 Preliminary Design Review (PDR) . . . . .	19
2.2.2 Critical Design Review (CDR) . . . . .	21
2.2.3 Coding and Debugging . . . . .	23
2.2.4 Formal Qualification Test (FQT) . . . . .	24
2.2.5 Functional Configuration Audit (FCA) . . . . .	26
2.2.6 Physical Configuration Audit (PCA). . . . .	27
2.3 Design Change and Error Correction Control . . . . .	27
2.3.1 Design Change and Error Correction . . . . .	27
2.3.1.1 Version Concept . . . . .	28
2.3.1.2 Design Control . . . . .	29
2.3.1.3 Error Correction Under Internal Configuration Management . . . . .	30
2.3.1.4 Updating Process . . . . .	31
2.4 Transfer and Turnover . . . . .	32
2.4.1 Configuration Management . . . . .	33
2.4.2 Change Processing During Transfer and Turnover . . . . .	33
2.4.3 Software Documentation . . . . .	34
2.5 Maintenance During Deployment Phase . . . . .	34



TABLE OF CONTENTS (cont'd)

	<u>Page</u>
SECTION 3. APPLICABLE REGULATIONS, SPECIFICATIONS, AND STANDARDS . . .	37
3.1 RSSs with Direct Impact on Software Maintenance . . .	37
3.2 Implementation of RSSs . . . . .	39
3.3 Potential Pitfalls in Applying RSSs . . . . .	41
APPENDIX A - Designing Maintainable Software . . . . .	43
APPENDIX B - Glossary . . . . .	57
APPENDIX C - Bibliography . . . . .	61

LIST OF FIGURES

Figure 1. Summary of System Acquisition, Model CPCI, Computer Program Life Cycle, and Command Responsibility . . . . .	6
Figure 2. Milestones for Acquiring Maintainable Software . . . . .	12
Figure 3. Cost of Approaching Hardware Capacity . . . . .	16
Figure 4. Software Errors Cost More to Fix as Computer Program Life Cycle Advances . . . . .	22
Figure 5. "Harmonics" Nature of Error Correction . . . . .	29
Figure 6. Design Approaches to Increase Module Independence . . . . .	45
Figure 7. Some Approaches for Defining Modules . . . . .	46
Figure 8. Top Down Development . . . . .	49
Figure 9. Summary of Coding Techniques . . . . .	51
Figure 10. Checklist for Commenting Computer Program Code . . . . .	52
Figure 11. Structured Programming Basic Control Structures . . . . .	54

## SECTION 1 - INTRODUCTION

### 1.1 PURPOSE

The Software Maintenance guidebook is designed to assist Air Force Electronics Systems Division Program Office personnel in the acquisition of maintainable command, control, and communications system software procured under Air Force 800-series regulations and related software acquisition management concepts. Many of the items and procedures discussed are applicable to smaller, less complex systems; but, in all cases, the guidance herein should be tailored to the needs of individual projects. The information provided in this guidebook is directed towards Program Office management personnel and a member of the Engineering Division, referred to as the Software Director, who is generally responsible for managing software acquisition.

### 1.2 SCOPE

The scope of this document is limited to those acquisition and development activities which impact software maintenance and which occur prior to the end of Full-Scale Development. It includes discussions of system turnover to the using command and the transfer of program management responsibility to the supporting command. The computer program life cycle is also considered. Its relationship to the Software Acquisition Management (SAM) cycle is shown in Figure 1. Most of the information provided in this guidebook covers the implementing command's responsibilities during the SAM cycle. However, software maintenance during the Deployment Phase is also discussed to provide the background for proper planning. Current programming concepts are discussed as well as the military regulations, specifications, and standards (RSSs). Within these constraints, this guidebook emphasizes what the Program Office (PO) can do to specify and procure maintainable software, including procurement of the facilities, support tools, and documentation necessary to support software maintenance activities. Guidance is given by:

- Defining software maintainability and those factors necessary to achieve maintainability (see 1.3).
- Locating those points in the acquisition cycle where the maintainability aspects of software can most appropriately be evaluated (see 2.2).
- Listing those considerations which should be made to identify support requirements, e.g., facilities, support software, and training (see 2.1).
- Defining the role of testing in the acquisition of maintainable software (see 2.2 and 2.3).

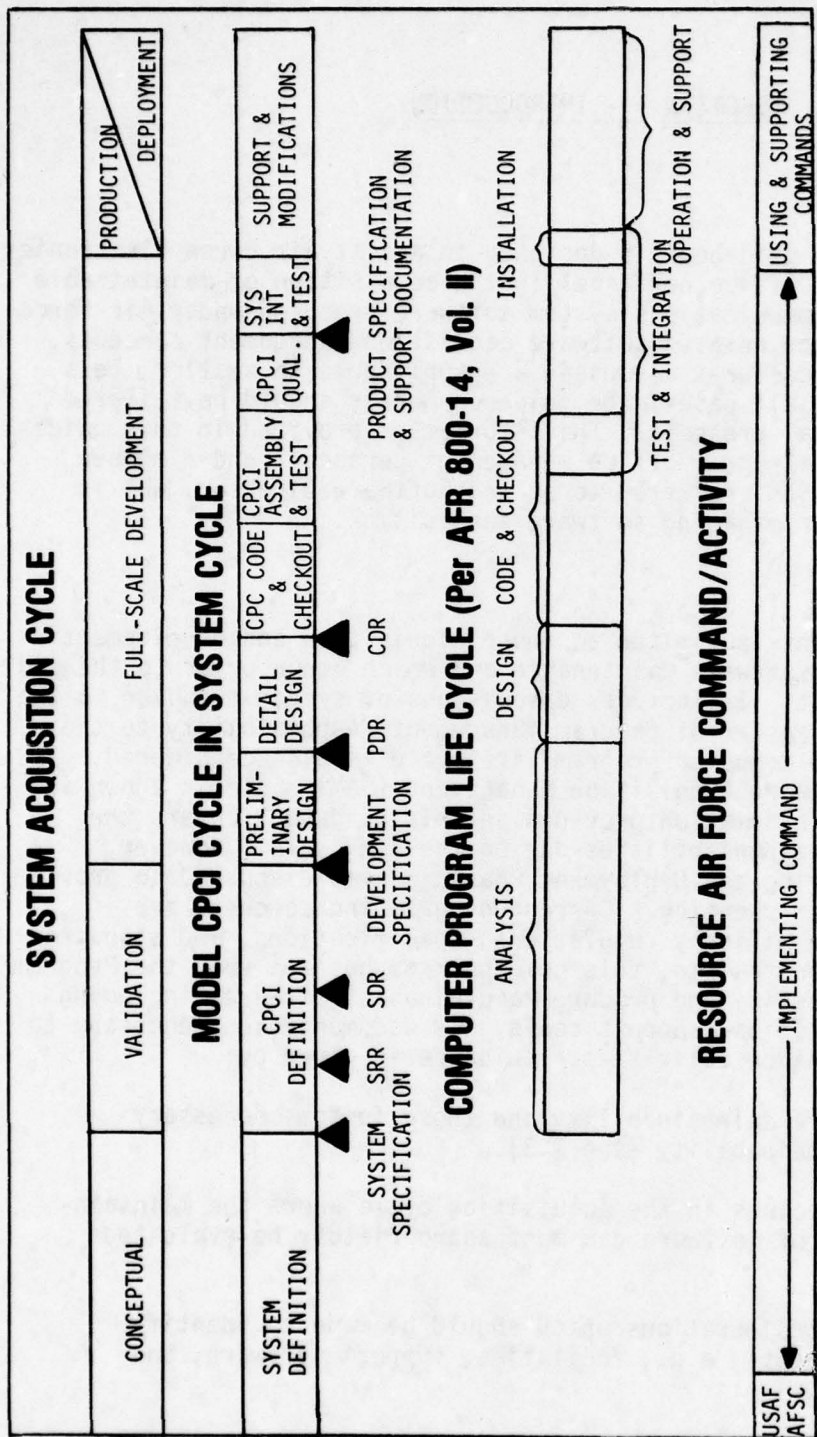


Figure 1. Summary of System Acquisition, Model CPCI, Computer Program Life Cycle, and Command Responsibility

- Outlining the relationship of the Computer Resources Integrated Support Plan (CRISP) to the acquisition process, to the Program Management Responsibility Transfer (PMRT), and to software turnover (see 2.1, 2.4, and 2.5).
- Describing the various concepts and tools which can aid in the acquisition of maintainable software (see 2.1 and 2.2).
- Discussing the significance of configuration management concepts to software maintenance (see 2.1 through 2.5).
- Describing the use of a Program Support Library (PSL), version releases, and Version Description Document (VDD) (see 2.2 and 2.3).
- Identifying pitfalls and practical approaches to developing maintainable software and controlling the software maintenance process (see 2.1, 2.2, 2.3, and 2.5).
- Discussing a strategy for handling software maintenance for multiple sites (see 2.5).

This guidebook does not elaborate upon the various types of funding citations or their use. However, the SD should be aware that (1) the implementing command is responsible for planning the necessary funds for development through the Full-Scale Development Phase, and (2) that these are normally R&D funds. After PMRT, the supporting and using commands are responsible for funding operational support and maintenance (Operations and Maintenance [O&M] funds). Therefore, the discussion in this guidebook, from Conceptual Phase through Full-Scale Development Phase, is concerned with activities which the SD can control. Subsequent to Full-Scale Development, this guidebook is concerned with activities for which the SD must plan.

### 1.3 SOFTWARE MAINTAINABILITY

The following discussion defines software maintainability and identifies the factors necessary to achieve maintainability.

#### 1.3.1 Definition

Computer-based command and control systems characteristically have a long life and usually must be adapted at one time or another to new operational requirements or to new equipment. In these events the computer program must be modified, or extended, perhaps drastically. In addition, although thoroughly tested prior to transition and turnover, large software systems

inevitably contain undiscovered errors which surface eventually and must be corrected. In this context, then, the following definition of software maintainability is adopted:

*Software maintainability is the property that, within a given operational environment, software can be corrected, amended, or modified to meet new requirements in a timely and cost effective way, and without regression of the parent system.*

The word regression in this definition refers to an unfortunate property in some large software systems that modifications to one portion may provoke errors in another. Changes therefore may lead to a domino effect requiring more changes and more testing. Some old, much modified systems or modules may actually die of regression.

### 1.3.2 Factors which Support Software Maintainability

Factors affecting software maintainability fall into three categories:

- Personnel, facilities, and tools.
- Documentation and configuration management.
- Inherent design and understandability.

Personnel, facilities, and tools comprise the maintenance environment, part of the operational environment caveated in the definition of software maintainability. Although this environment is the responsibility of the maintenance organization, the acquisition organization must understand it well enough to account for its essential needs prior to turnover. For personnel, consideration of the qualifications of the maintenance programmers, whether entry level, blue suit, intermediate, or system level may influence the choice of programming language. For instance the automatic test equipment that is part of a large radar early warning system should be programmed in ATLAS, an application-specific language which allows knowledgeable hardware people who do not have specific programming experience to construct new tests. If a more general, procedure-oriented language such as FORTRAN is used, more expensive training requirements will be generated that will endure for the life of the system. Issues such as this should be addressed by the PO in conjunction with the Computer Resources Working Group (CRWG), and necessary analyses and trade-offs performed and referenced in the CRISP.

Facilities, as opposed to tools, are existing facilities within which software maintenance will be performed. If they are different from the (contractor's proposed) development environment, then the Computer Program Development Plan (CPDP) should address the resultant problems of transporting or reacquiring essential tools.

The term tools, refers to software tools including compilers, PSL, event simulators, language and machine simulators, environment simulators, test generators, and data reduction processors. The PO must acquire the ownership and documentation of the tools needed for maintenance, must control the contractor's use of proprietary tools, must negotiate limited rights where appropriate and in general, assure that needed tools are available and usable for maintenance. The Request for Proposal (RFP) should require identification of all proprietary tools and any conditions or restrictions placed upon their use. The contract should reflect this and should require delivery, or contain options to purchase all needed tools. The contract should contain, or at least should not preclude, any needed licensing agreements.

Documentation and configuration management provide the necessary technical and status information to support software maintenance. Documentation includes the specifications and supporting materials, such as positional handbooks and user guides, needed to modify the software. The documentation must be easy to use, understandable, and, most importantly, must reflect the current software. Implicit in every software maintenance task is the requirement to keep the documentation up-to-date. Careful definition and organization of documentation with consideration of its intended uses and requirements for update can significantly ease the software maintenance task.

Configuration management includes the identification and statusing of all Configuration Items. Accurate statusing of the software, documentation, and all changes, both proposed and installed, is a must for effective software maintenance (see Configuration Management guidebook).

Design of the software is the key to its maintainability. The top-level of the design is specified in the Development (Part I) Specification in terms of Computer Program Configuration Item (CPCI) definition, performance requirements, interfaces, growth requirements, and design constraints. Maintainable software requires adequate design. This leads the PO to place design constraints on the developer. In certain critical areas of the software design, these constraints may limit performance and they may therefore have to be relaxed. For example, the developers may be required to use a High Order Language (HOL) for programming which cannot be accommodated in the hardware specific and time-critical portions of the executive. Most of the design does not appear in the Development Specification; it is documented in the Product (Part II) Specification. Proper design includes:

- A limited number of interfaces between modules.
- Communication between modules limited to the defined interfaces.
- Well documented, easy to understand design.
- Limited equipment interfaces.
- A controlled data base.
- Limited access to the data base by each module.
- Programming style for clarity of function (readability) and ease of verification.
  - Singular functional performance of each module which leads to small modules (i.e., only one function per module).
- Separate modules for input, output, and computation of functions.

#### 1.4 CONTENTS

The subsequent contents of this guidebook are presented in two sections and three appendixes, as follows:

- Section 2 - Acquiring Maintainable Software. Discusses: (1) the definition and specification of maintainable software; (2) monitoring the evolving software design and evaluating contractor effectiveness; (3) design change and error correction during subsystem-development test and evaluation; (4) transfer and turnover; and (5) maintenance during the Deployment Phase.
- Section 3 - Applicable Regulations, Specifications, and Standards. Discusses those regulations, specifications, and standards that impact the development of software.
- Appendix A - Designing Maintainable Software. Discusses considerations important to the design of maintainable software. Provides information that the PO can use to evaluate the contractor's design as defined in his proposal and CPDP. Also provides information that the PO can use in preparing for PDR, CDR, PCA.
- Appendix B - Glossary. Defines specific terms and acronyms used in this guidebook.
- Appendix C - Bibliography. Provides a list of books and papers that are particularly relevant to the subject of software maintenance.

## SECTION 2 - ACQUIRING MAINTAINABLE SOFTWARE

This section discusses how the SD can plan and monitor the computer program life cycle to obtain maintainable software. It provides specific guidance for the SD during software acquisition and development. Figure 2 relates the contents of this section to the major milestones of the system acquisition cycle.

### 2.1 DEFINING AND SPECIFYING

Planning for maintainable software should be started during the Conceptual and Validation Phases. The SD is responsible for ensuring that planning for maintainable software and support requirements are considered in the analyses and tradeoff studies conducted during these phases. Such requirements include excess computer capacity, support software, and documentation. These requirements must be reflected in the contractual specification, the Statement-of-Work (SOW), the Contract Data Requirements List (CDRL), the Data Item Description (DIDs), and the Computer Resources Integrated Support Plan (CRISP). The SD should monitor studies and tradeoffs at the System Requirements Review (SRR) and System Design Review (SDR) to ensure that both software maintenance and mission performance requirements are being satisfied.

#### 2.1.1 Planning Considerations

Planning for maintainable software is a relatively new concept and currently available guidance is minimal. During initial planning, the SD in consultation with the using and supporting commands should determine the scope, content, and level of maintenance required to satisfy their needs. These needs should be documented in the CRISP. See the Software Development and Maintenance Facilities guidebook for more information regarding planning considerations. The software maintenance needs should be determined by the PO, the user, and the maintainer, based on tradeoffs, studies, and analyses that evaluate life cycle cost impacts. The following areas should be evaluated as quantitatively as possible:

- Contractor vs In-House Support. This decision will impact training, documentation, facilities, personnel, housing, computer resources, and data rights. Even if contractor support is chosen, the SD has to determine where maintenance will be performed and what deliverables will be required, whenever possible. It is wise to insist upon unlimited data rights, full documentation, no proprietary software, and full delivery of all source and object code and all software test tools developed under the contract. Cost is obviously a major consideration, but initial development costs must be balanced against overall life cycle cost. This allows the Government the option of either procuring maintenance support from industry or developing an in-house capability at a later time. It should also be determined at this time what software the using and support commands will be responsible for maintaining.



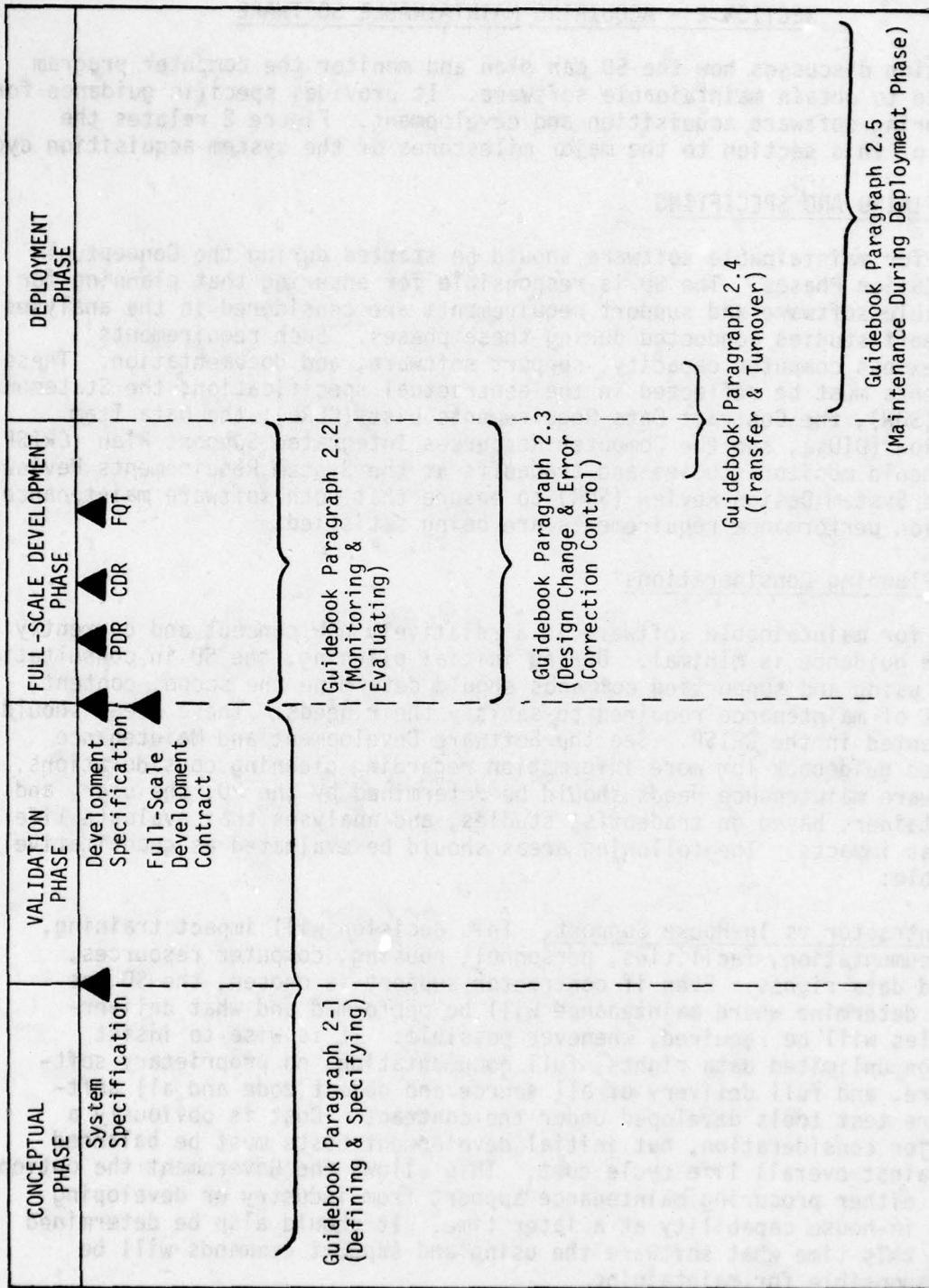


Figure 2. Milestones for Acquiring Maintainable Software

- Computer Resources. An initial estimate of the resources required to develop and maintain the software is needed for funding considerations, to guide the contractor, and to evaluate his proposal. The estimate should be based on a tradeoff among at least the following alternatives based upon performance and availability requirements: a dedicated support facility, dedicated time on the operational system, dedicated time on a backup system, or time sharing in a multiprocessor and/or multiprogramming environment. Cost of equipment, facilities, trained support personnel, and special software should be considered along with the required responsiveness of the maintenance organization.
- Special Support Software. Based upon software maintenance requirements and the class of computers being considered, the SD should evaluate the need for simulation, recording, and data reduction tools. An estimate should be made of additional utility/support tools such as debugging aids, COMPOOL generators, and compilers or assemblers. A PSL should also be required and any special software needed to support the PSL should be specified. See RADC-TR-74-300, Volume VI, for a discussion of the PSL. All special support software, if not available off the shelf, should be scheduled for early development, since the development of the operational software is dependent upon their availability. If a compiler is required, it should be developed to a specified standard and qualified before it is used for program development. New compilers can be a high risk area and should be scheduled early in the development cycle, when staffing is low, so that delays will not have significant cost impact. Support software, which is properly planned, developed, and qualified, will enhance both development and maintenance of the operational software.
- High Order Language. AFR 300-10 requires the use of a standard HOL. JOVIAL is the one most likely to be required. In general, the compiler should be acquired and qualified prior to the start of coding. Currently, ESD can obtain a JOVIAL (J3) compiler in 6-to-9 months, using the Rome Air Development Center (RADC) JOVIAL Compiler Implementation Tool (JOCIT). Soon, a J73/1 compiler should be obtainable in the same time period. Use of the JOCIT tool limits the risk and cost of these compilers. ESD also has a JOVIAL Compiler Validation System (JCVS) which is used to assure that the compiler meets its specification. Any J3 or J73/1 compiler should be certified using JCVS.

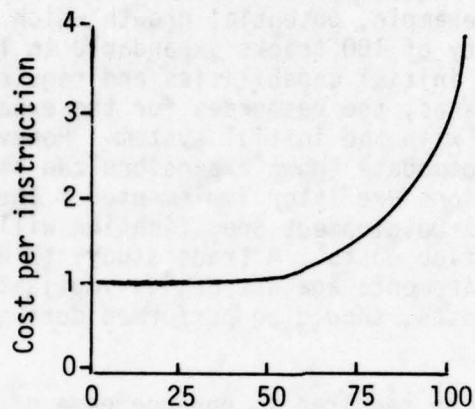
- Development Schedules. Most development schedules for large systems in recent years have been too short. As a result, in the rush to complete the project, both software development and maintenance considerations suffered. Development (Part I) Specifications are usually required from 90-120 days after contract award. The original intent of this time requirement was to update the Part I Specifications developed during the Validation Phase. However, on many occasions, the Part I Specifications were initiated and developed during this time. When this occurred, the specifications were generally incomplete and inconsistent. It is better to spend the equivalent time on fully determining the performance requirements (usually a 6- to 9-month effort) before writing the Development (Part I) Specification. The SD must assure adequate time in the project schedule for the contractor to design and produce the embedded software. The SD should require that CDRs and PQTs be held; however, he should let the contractor propose the schedules in a way that best supports the contractor's approach. Emphasis should be on the adequacy of the overall design before the detailed design is started. Good design is a major contributor to maintainable software.
- The Computer Resources Integrated Support Plan. The CRISP identifies responsibilities for the management and technical support of computer resources, including responsibilities for maintenance. The initial plan is developed during the Validation Phase. The specific information to be supplied in the CRISP is presented in AFR 800-14, Volume II, Chapter 3. The CRISP evolves as the project evolves and requires coordination with the using and supporting commands to assure that their support concepts for computer resources are properly reflected. The coordinated planning for computer resources which is documented in the CRISP, must establish the configuration management, resources, documentation, funding, scheduling, integration, training, support software, facilities, and provisions for transfer and turnover, which will enhance software maintenance.

#### 2.1.2 Development Specifications and the Full-Scale Development Phase RFP

It is important to ensure that software maintenance requirements are specified in the Development (Part I) Specification and reflected in the contractor's Full-Scale Development Phase proposal. The Development Specifications and the SOW should reflect the results of trade-offs and planning done in the Conceptual and Validation Phases and include requirements in the following areas:

## DEVELOPMENT SPECIFICATIONS

- Known software growth requirements should be specified to facilitate future upgrade. For example, potential growth which impacts design (e.g., initial capacity of 100 tracks expandable to 150) should be specified in terms of initial capabilities and requirements for expansion. In many cases, the resources for the expanded capabilities need not be purchased with the initial system. However, design of the initial system to accommodate known expansions can save both time and money when the expansions are later implemented. The SD should be aware that too broad a Development Specification will result in higher design and implementation costs. A trade study, to evaluate potential growth requirements against design implications and development and life cycle costs, should be performed during the Validation Phase.
- A standard HOL should be required to enhance ease of maintenance with a minimum use of assembly language allowed in areas where code efficiency or machine dependency (e.g., I/O) require it. The contractor should identify and justify all areas where assembly language is required.
- Requirements for programming standards should be specified.
- All necessary support software should be functionally specified. This should include a requirement for a minimum capability as defined in RADC-TR-74-300, Volume VI.
- Excess computer capacity should be required to allow software growth and error correction. Roughly, the initial equipment capacity should be twice the initial sizing and timing estimates. To improve system potential over a long life cycle, the computer system should be expandable to several times its initial capacity (see Figure 3).
- The Development (Part I) Specification is the most appropriate contractual tool for the acquisition of maintainable software. In addition to the CPCI performance requirements listed in paragraph 2 of the Development Specification, the Special Requirements paragraph 3.2, provides a format for specifying the following types of software maintenance requirements (see MIL-STD-483, section 60.4.3.2.2 for a description of paragraph 3.2):
  - The use of programming standards
  - Specific program organization requirements
  - Program design considerations which ease modification
  - Expandability (growth potential) requirements



Percent utilization of time and core

Figure 3. Cost of Approaching Hardware Capacity\*

FULL-SCALE DEVELOPMENT PHASE RFP

- The SOW should specify that software maintenance requirements will be reviewed at Preliminary Design Review (PDR) and Critical Design Review (CDR). The reviews should allow the contractor to use functional flow diagrams, structured charts, or other presentation techniques to present system and program-level design detail. See the Reviews and Audits guidebook for further guidance regarding PDR and CDR.

\*From "Through the Central 'Multiprocessor' Avionics Enters the Computer Era" (see Appendix C).

- The SOW should require that a Computer Program Development Plan (CPDP) be submitted with the Full-Scale Development Phase proposal.\* It should be updated periodically to reflect the evolving software development plans. This plan covers the complete software development efforts. Much of the contractor-development methodology described in the CPDP can significantly impact software maintainability. For example:
  - Plans to deliver the PSL, including support software, contents, and procedures.
  - Applicability of the contractor's internal configuration management plan to include: change status reporting, control procedures problem reporting, and error correction procedures.
  - Plans for updating test plans and procedures to accommodate ECPs and assure continued applicability during Deployment.
  - Top-down design and structured programming approach.
  - Data naming conventions and data base control procedures.
- If in-house software support is selected, then a training program should be requested. This allows the PO to specify training materials and set the necessary schedules to assure that support documentation is available at System Development Test and Evaluation (DT&E). Since most C<sup>3</sup> systems are unique, the development contractor should be required to present a training course in sufficient time to support System DT&E. This should allow maintenance programmers to get on-the-job experience before they take over maintenance responsibility.
- The CDRL should specify the software maintenance features of the deliverable documentation, i.e., commented listings. The standard DIDs should be modified to meet the individual needs of the program. For additional guidance see Software Documentation Requirements guidebook.

---

\*The CPDP may be initially prepared during the Validation Phase. It must be updated at the start of Full-Scale Development.

- The contract Work Breakdown Structure (WBS) should reflect all support requirements associated with acquiring maintainable software including unique identification of all support software, documentation, training, extra equipment, and facilities related to maintenance. These requirements should be shown down to the deliverable contract line item/configuration item level (see SOW Preparation guidebook, Appendix A). This is necessary to properly identify and monitor the cost associated with acquiring maintainable software and to have some options for deleting capabilities if there are schedule or budget constraints.

The Full-Scale Development Phase RFP should require the contractor to state in his proposal how he will design the software for ease of maintenance. Requirements for modular design, top-down design, or structured programming must be defined in detail to be meaningful. Generally this information is included in the CPDP. Until specific design techniques such as these have been successfully demonstrated on C<sup>3</sup> systems it is better to leave their definition and implementation to the contractor's discretion.

There is much that the contractor can do to enhance or obstruct the development of maintainable software. His proposed technical approach, management approach, and CPDP should be evaluated to determine whether he has:

- Tailored his proposed documentation to support software maintenance.
- Developed design and coding standards to support software maintenance.
- Shown how his development support software will be applicable during the Deployment Phase.
- Included software maintenance considerations in all aspects of his design, code, test, and documentation.

The following pitfalls should be avoided:

- Development plans that do not include time or budgets to rigorously design the software to include maintenance considerations.
- Off-the-shelf software that is difficult or impossible to maintain.
- Contractor-proprietary tools used during development but unavailable (although needed) during Deployment.

## 2.2 MONITORING AND EVALUATING

During the Full-Scale Development Phase, the SD is concerned with monitoring the evolving software design and development tasks and evaluating how effectively the contractor is meeting the specified contract requirements. To do this, he employs the techniques of quality assurance, configuration management, technical and management reviews, and verification and validation. These subjects are discussed in other volumes of this guidebook series (see Preface for list of other guidebooks). The intent of this discussion is to assist the SD in acquiring maintainable software by providing checklists to supplement the normal review process.

### 2.2.1 Preliminary Design Review (PDR)

The purpose of PDR is to evaluate and monitor the progress and technical adequacy of the selected design approach for each CPCI. The review should emphasize design, language usage, and programming standards. Although the PDR is a design review, it should also be used to review the contractor's planned implementation methods, as described in the CPDP. The SD should check for the following features which facilitate the development of maintainable software:

- Has the CPCI been designed in a manner that provides for ease of modification, as planned for in the CPDP?
- Have all needs to deviate from the design approach of the CPDP been identified and coordinated? (For example, real-time requirements may dictate other than a top-down approach.)
- Have CPC and data base interfaces been defined so that independent detail design can be started at a lower level? Have interfaces been defined in a simple and explicit manner?
- Have functions and subfunctions been allocated to CPCs in a way that enhances modularity and functional independence?
- Has the CPCI data base been defined in a symbolic manner?
- Is there a centralized data-definition capability, such as a COMPOOL? If not, is there a procedure established to define and control the data base definitions?
- Have all areas where assembly language is required been identified and justified?



- Are programming standards with coding examples available to the programming staff? Do the standards cover techniques for developing a modular and structured CPCI? Have methods for improving module independence been included? (See Monitoring and Reporting Software Development Status guidebook, Appendix II, pages 64-67.) Have procedures been established for enforcing the standards (e.g., program walkthroughs and code audits)?
- Have programming personnel been trained in the concepts of top-down implementation, structured programming, operating-system requirements, library procedures, and modular-coding techniques? If not, is a training program scheduled for all current and newly-assigned personnel?
- Is a standard HOL used? If not, is the contractor HOL selection based on cost or technical considerations?
- Will the compiler be qualified before it is required for coding?
- Have all performance requirements been allocated to CPCs?
- Have support tools been defined? Have those tools that require new development been designed? Have they been designed in a modular manner? Have debugging tools been defined? Have all modifications to commercial off-the-shelf and Government-Furnished Equipment (GFE) debugging aids been identified? (Government ownership of support and related documentation should have been established in the contract, but should be reviewed at this time.
- Are the test requirements for maintainable software included in the updated CPCI test plan?\* In general, maintainable software design features should be reviewed by inspection.

The SD must take sufficient time to prepare and insist upon sufficient technical resources to adequately review the materials presented at the PDR. The technical reviewers must be capable of evaluating the material presented by the contractor and be familiar with the Development (Part I) Specification.

---

\*The original CPCI Test Plan should have been submitted with the Full-Scale Development Phase proposal.

If it is determined that the contractor is not sufficiently prepared then the PDR should not be conducted until the contractor is ready. Maintenance-related problems which may be observed at PDR include CPC or data interfaces which are too complex, modularity which may not be apparent, or data-base design which may not include considerations for change. The SD should identify such problems and ask the contractor to resolve them. It is better to delay the project at this point to get a design that is maintainable and meets all performance requirements (when contractor staffing is at a lower level) rather than to wait until CDR or Formal Qualification Test (FQT) when delays are more expensive. See Figure 4 which illustrates the increasing cost of correcting errors as the development progresses; the system becomes more complex and the number of people delayed increases.

### 2.2.2 Critical Design Review (CDR)

The purpose of CDR is to provide a formal technical review, or series of reviews, at completion of the detailed design of each CPCI, or group of related CPCs in a large, complex CPCI. Successful completion of CDR signifies verification of the detailed CPCI design and allows initiation of CPC code and test activities.

From the design detail available at CDR, the individual program modules, their interfaces, and associated data base requirements can be identified and coded.

The SD's emphasis during CDR should be on the program structure, modularity, language usage, programming standards, support tools, data base design, interfaces, and planned coding techniques. A traceability matrix should be available that further relates the requirements directly to the implementing modules (as opposed to CPCs that were reviewed at PDR). In reviewing the contractor's design at CDR, the SD should check for the following additional features which facilitate the development of maintainable software:

- Have all software modules been specified? If an incremental CDR, have all modules for this build been identified? (See Appendix II of the Monitoring and Reporting Software Development Status guidebook.)
- Are all module interfaces defined and documented in accordance with the CPDP? Is all control data passed only through the defined interfaces? Has the amount of interface data been minimized?

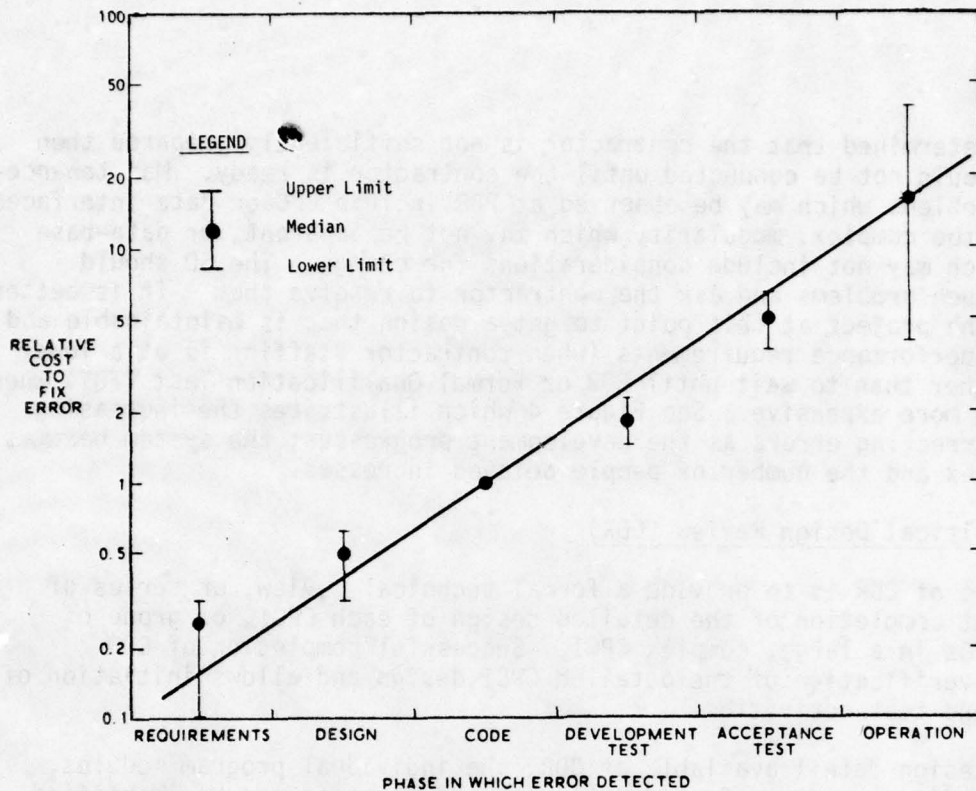


Figure 4. Software Errors Cost More to Fix as Computer Program Life Cycle Advances\*

- How well have the modules retained their independence?
- Have machine dependencies been isolated and encapsulated in accordance with the CPDP?
- Has the system data base been designed and documented? Has it been symbolically defined and referenced? For example, was a COMPOOL used?
- Is I/O centralized and separate from computation functions?

\*Figure adapted from Boehm, "Software Engineering," see Appendix C.

- Are module source-code estimates within the contractor's module-size limitations?
- Are modules functionally cohesive (i.e., limited to a single or small number of closely related functions)?
- Have all modules requiring assembly language been identified and justified?
- Have all support tools needed for coding and debugging (e.g., pre- and post-processor) been produced? If not, are they scheduled early enough to meet the needs of the development schedule?
- Have all modules been designed to have single entry and exit points (with the exception of certain computer interrupts and erroneous condition exits)?
- Whenever initialization or housekeeping is required, will those functions be internal to the module requiring them?
- From a recovery point of view, computational or I/O modules should not abort. They should pass an error condition back to the main control level which is designed to make abort or recovery decisions. Has this concept been followed?
- Does the PSL reflect the software structure? Have procedures and libraries been established to control the baselined source and object files and listings as they are produced?
- Do the test procedures provide for inspection of maintainable software requirements (e.g., module size, structured code, HOL and assembly language, and module independence)?
- Will a code auditor be available at compile time to monitor software characteristics and enforce standards?

### 2.2.3 Coding and Debugging

During the process of coding and debugging, the contractor is developing and testing individual modules in accordance with their design. If an incremental build or top-down implementation approach is used, the modules are tested in execution order (i.e., integrated in their final structure and tested with minimal need for test drivers). In most complex systems, a combination of top-down and bottom-up implementation is used. The reason this must be permitted is that some existing software may be used or adapted for use in the new CPCI. In addition, the coding on some modules may start earlier than on the CPCI as a whole. Considerable savings in time and dollars may accrue from

from a mixed top-down/bottom-up approach. When bottom-up implementation is used, it is especially important that interfaces be rigidly defined, maintained, and adhered to. The approach taken should be proposed by the contractor. The SD may gain visibility into the development process during the contractor's Computer Program Test and Evaluation (CPT&E) or the SD can use the Preliminary Qualification Tests (PQTs) to verify if maintainable software requirements are being met. To facilitate the development of maintainable software, the SD should check for the following:

- Review PSL content to see if module size and program structure match the design.
- Review the contractor's planned development methods, procedures, and standards, as documented in the CPDP, and ensure that they are being followed.
- Select modules and review their code to see if coding standards are being enforced, or review code auditor output, if available. This should be done in conjunction with the contractor's QA manager and in accordance with the contractor's QA plan.
- Review the current data base to see if it meets its specification and is symbolically defined.
- Verify that the necessary development and debugging tools are available and are being used.
- Verify the language used by each module against its design documentation and against the coding standards. Ensure that assembly language is not embedded in the code unless it was explicitly called for.
- Review the code to ensure that the following difficult-to-maintain features\* have not been included:
  - Self-modifying code
  - Absolute addressing
  - Embedded constants and literals
  - Relative addressing

#### 2.2.4 Formal Qualification Test (FQT)

During Subsystem DT&E the contractor conducts FQT(s) (see Verification guidebook) of the CPCI(s) under development.

\*Normally these problems are associated with assembly language programming.

If standards of modularity, language, and independence of modules, are not met, it is difficult to take corrective action at this time without significant cost and schedule impact. Thus, it is important to assure adequate design and monitor its implementation through formal reviews and PQTs. Corrective action should be taken prior to FQT because the contractor still has programmers available and is trying to meet FQT milestones. Again, using PQTs to inspect the code of selected modules as they are being developed minimizes the risk.

The following items should be inspected primarily at PQT but also at FQT for maintenance implications:

- Do the test procedures call for adequate inspection of the specified maintainable software attributes? Check module size, language, structured code, adherence to programming standards, and code readability.
- Have design changes, requirement changes, and error corrections caused major impacts on the software structure?
- Have the traceability matrix [in the Product (Part II) Specification] and the test procedures been updated to reflect design and requirements changes? (See the Verification guidebook.)
- Have those portions of the software that are time-sensitive been identified and documented? Have those portions of the code been adequately commented to alert maintenance programmers?
- Are the listings readable and reasonably self-documented?
  - Are they adequately commented?
  - Can they be easily reviewed?
  - Is it clear what each area of code is intended to do?
  - Are the data for references symbolic and are they meaningful?
  - Are the date and version of the listing compatible with the contractor's list of materials to be qualified?
- Have all development and test support tools been found acceptable? Make sure the contractor qualifies all tools to be delivered.

NOTE

*PQT and FQT plans and procedures are directed at testing CPCI performance against the Development (Part I) Specification. Comments from review of program listings can be transmitted to the contractor at PQT and FQT with notice that unless improvements are made, the Product Specifications (source listings) will not be accepted at PCA. However, official response should be delayed until data delivery at PCA.*

### 2.2.5 Functional Configuration Audit (FCA)

The purpose of the FCA is to verify that the CPCI's actual performance meets Development Specification requirements. Of particular interest to software maintenance considerations is the contractor's briefing for each CPCI [see MIL-STD-1521A(USAF), paragraph 50.4.12a]. At the briefing, the contractor provides a general presentation of the entire development test effort, delineating problem areas as well as accomplishments. The briefing should include an account of the ECPs incorporated and proposed and the contractor should identify any Development Specification requirements that he was unable to meet, including a proposed solution to resolve any CPCI inadequacies caused by not meeting the requirements. When using the briefing to promote maintainable software, the SD should notify the contractor that the briefing information will be evaluated for maintenance considerations. In particular, the SD should:

- Determine the planned disposition of the ECPs not yet incorporated and evaluate their probable impact on software maintainability.
- Evaluate the causes and solutions of the problems which occurred and determine whether similar problems are likely to occur as changes or error corrections are installed during Deployment.  
If so:
  - Can additional methods or procedures be devised to alleviate the future problems?
  - Should new or modified training courses be required?
  - Is any additional support documentation required?

#### NOTE

*The evaluation of software maintenance requirements is a continuous process. Affirmative answers to any of the above questions may require additional funds. In any case, the command responsible for software maintenance should be notified of the potential problems and of the recommended actions.*

### 2.2.6 Physical Configuration Audit (PCA)

The PCA is designed to verify that the product baseline and associated material released for System DT&E is compatible with what was qualified at FQT. This process verifies that the source and object code listings and documentation are compatible. The PO and the contractor conduct the audit. At completion of PCA, the Product (Part II) Specification is baselined. A Version Description Document (VDD) is delivered to define the content of the version released for further System DT&E testing. The availability of a PSL should help expedite this process since the PSL maintains the integrity of the source and object code and associated listings. Reports from the PSL and a review of the program description against the program listings and data base should be sufficient to verify the integrity of the product. The SD should require that the contractor provide source materials (a listing of the source code, supporting documentation, and the object code in machine-readable format) for all products audited. Software cannot be maintained without program source materials. If commercial off-the-shelf programs and support tools are used to develop the programs and are required for operations and maintenance, then the SD must assure that the items and supporting documentation are available without restrictive data rights. It might be cheaper in terms of life cycle cost to develop new support programs if this is a problem.

### 2.3 DESIGN CHANGE AND ERROR CORRECTION CONTROL

Although maintenance does not officially start until the Deployment Phase (after Program Management Responsibility Turnover (PMRT)), the need to correct and modify software begins during Subsystem DT&E. Many of the software maintenance practices, procedures, and tools used during Subsystem DT&E will continue to be used throughout the life cycle. The contractor places the software under the control of his configuration management procedures at this time. These procedures are normally identified in the contractor's configuration management plan.

During Subsystem DT&E, design changes, not affecting the Development (Part I) Specification, and all error corrections are under the contractor's control. Changes affecting the Development Specification require ECP action. However, the basic approach to making changes to the configuration-controlled software should be the same, whether contractor-initiated or the result of an ECP (see Configuration Management guidebook).

#### 2.3.1 Design Change and Error Correction

Design change and error correction is a natural part of the software development process and is present throughout the Full-Scale Development Phase. They must be accommodated by the contractor's design control and configuration management procedures. All changes must be documented and controlled to assure the integrity of the software design and performance. It is essential that every change be evaluated for schedule impact and consideration given to packaging



changes for release within a subsequent version of the computer program unless the change is critical to the continuation of qualification testing.

Design change costs vary, depending on the time of their occurrence during the computer program life cycle. As the development effort progresses, every change must be reviewed on the basis of need. This is especially true for changes to established interfaces or to the data base. All design change recommendations should continue to be documented, but implementation of some may have to be deferred until after the CPCI passes FQT.

#### 2.3.1.1 Version Concept

A version is the actual configuration of a CPCI which is introduced into the system for installation, test, or operation. The version concept allows the contractor to schedule a CPCI release for his internal testing with a specific set of capabilities or a specific set of changes. For example, using a top-down implementation scheme, version releases can be scheduled and tested to reflect the hierarchical development of the CPCI. After development and internal test of the CPCI is completed, it can then undergo qualification testing. The version concept is also used to package modifications associated with ECPs. (See Verification guidebook for a more detailed description of contractor internal testing. See Appendix II of the Monitoring and Reporting Software Development Status guidebook for more information on version implementation.)

The version concept allows the contractor to better schedule the development effort and to assure that the necessary support tools, documentation, and test procedures are available to support the developing CPCI. This same approach holds true for modifications (i.e., ECPs). Instead of the baseline continuously changing, modifications can be scheduled, developed, and tested as a group. This improves control, allows development to proceed in parallel with qualification of the previous version, and provides for scheduling of the support products and documentation. Figure 5 shows how the version concept can be used to localize errors within a specific level of the program. With the design of versions limited to specific software incremental areas, the errors reported with each new version should be limited to the scope of that version. The total number of errors should decrease as versions are implemented.

Since a PSL allows for multiple libraries, it provides a convenient mechanism for implementing a version concept [i.e., a new version of a program can be produced in a development library (or libraries) while the baselined version is tested]. A version identification (for each module), that changes when the code changes, also enhances configuration control. A PSL should allow automatic changing of the module's version identification when the code is changed.

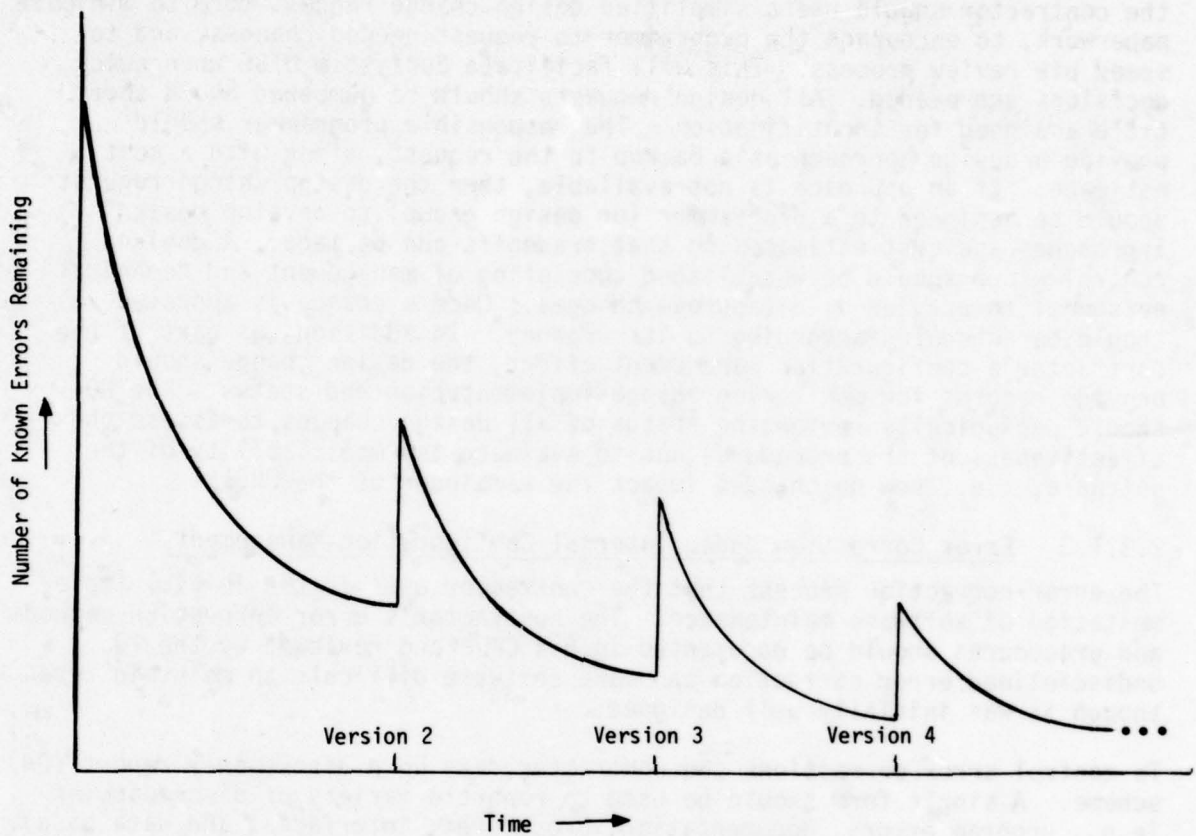


Figure 5. "Harmonics" Nature of Error Correction

#### 2.3.1.2 Design Control

A module's code should be placed under the contractor's configuration control following completion of CPT&E. At this point, it should be moved from the development library file to a controlled library file, and all additional changes approved by the contractor's configuration management. In top-down development, the code has been executed in the current operational structure and its stub replaced at this point.

When changes to the Development (Part I) Specification are required, the contractor should use a simplified design change request form to minimize paperwork, to encourage the programmer to request needed changes, and to speed his review process. This will facilitate Subsystem DT&E when quick decisions are needed. All design requests should be numbered and a short title assigned for identification. The responsible programmer should provide a design approach as a backup to the request, along with a cost estimate. If an approach is not available, then the design change request should be assigned to a programmer (or design group) to develop design approaches and cost estimates so that tradeoffs can be made. A design control group should be established consisting of management and technical personnel to provide or disapprove changes. Once a change is approved it should be scheduled according to its urgency. In addition, as part of the contractor's configuration management effort, the design change should provide records for monitoring change implementation and status. The PO should periodically review the status of all design changes to assess the effectiveness of the procedures and to evaluate the modifiability of the software, i.e., how do changes impact the remainder of the CPCI?

#### 2.3.1.3 Error Correction Under Internal Configuration Management

The error-correction process that the contractor uses is the initial implementation of software maintenance. The contractor's error correction methods and procedures should be documented in his CPDP and reviewed by the PO. Undisciplined error correction can make software difficult to maintain, even though it was initially well designed.

To control error corrections the contractor sets up a discrepancy report (DR) scheme. A single form should be used to report a variety of discrepancies (e.g., program errors, documentation, procedures, interfaces, and data base). Such a form minimizes confusion, can be used to document design problems, and can be used as an input to the design change process.

Once a DR is initiated, it should be evaluated by the contractor's program management and QA organization, logged for configuration management, and assigned to the responsible agency for correction as required. If the DR is a design problem, it should be sent to the design control group for resolution. If it is a program error, a correction must be prepared and the program module updated in a development library, and then tested and released for integration into the controlled library. The contractor's test director should determine when this correction will be integrated, based upon test schedules. Once it has been tested in its operational environment, the DR should be closed and the status updated. The PSL is of major importance to this process. It provides for multiple libraries, program identification updating, control (source code, object code, listings), and utilities to move files from one library to another, and provides reports on the contents of the library. The PSL, thus, can provide for strict configuration control. These same tools and procedures can be used by the maintenance organization during deployment.

#### 2.3.1.4 Updating Process

Future software maintenance requires that an orderly updating and changing scheme be implemented early in the development cycle. This process must ensure that every change is documented and that a consistent numbering scheme is used to relate the source listing, the supporting documentation, and the status records for each change. The process of correcting or changing a computer program during test involves a number of considerations and tradeoffs, including:

- Patching vs Symbolic Updating and Recompiling the Module. In some systems a patching capability allows the programmer to change specific instructions without assembling or compiling the module again. This approach allows a quick change to a localized area of the module and has been used on many projects. However, it is hard to control because there is often no direct correspondence between each computer program statement used in the patch and each statement used in the recompilation. Further, whenever patching is used during system test, it must be followed by recompilation and some repeated testing. Where source code is written in a HOL, it may not be possible to duplicate the binary patches, hence, regression may occur. Assembly language computer programs, although less desirable in other respects, are much safer to patch than HOL programs. With current computer technology, support/utility programs, and a PSL, patches should not be allowed. All corrections should result in updating the source code and recompiling the program. This will assure agreement between all representations of the program code (source code, object code, and listing), simplify configuration control, and eliminate a later clean-up program. All changes (alters) to the source code should have their DR or change control number inserted so that changes can be easily identified and cross-referenced back to the problem description.
- Alters vs a Complete New Program. In correcting or changing a program's (or module's) source code, it is possible for the programmer to submit a few changes (alters) or to submit completely new source code. During qualification testing, only alters should be accepted and, to enhance configuration control, these should be applied by the librarian against the source code. If completely new source code is provided, the programmer may try to include additional changes in addition to those fixing the DR. If found, additional testing will be required. If not found, then aberrations may show up later in a program that was thought to be thoroughly tested and qualified. If the change is so extensive that new source code must be released, then a comparison of the new object code and source code against the old object and source codes should be run and all changes identified. This approach may require the use of a utility program. It is desirable when updating the source code to retain the previous copy for recovery

purposes. In some systems, the last two copies are retained, and in others, all copies are retained. Again, a PSL should allow the contractor to maintain the required level of program archives.

- Document Maintenance. As software corrections are made, it is important to update the affected documentation to keep it current. This includes updating user guides, data base descriptions, and the program listings. The majority of changes will normally not affect any documents other than program listings and the data base. If detailed flow diagrams are included in the Product (Part II) Specification, as currently required by MIL-STD-483 and MIL-STD-490, then considerable effort will be required to update them. It is recommended that detailed flow diagrams not be procured. Instead, the effort should be spent on keeping the listings updated. This requires that comment and indentation standards be followed as corrections are made. This is especially important for comments since a partial or out-of-date comment may be misleading. As much of the descriptive documentation as possible should be kept in the PSL (especially if it has a text editor) to ease the update problem. The contractor should be required to keep his documentation up to date to prevent a large cleanup effort after Subsystem DT&E when the contractor may not have enough knowledgeable programmers assigned to the program. This will ease the work prior to delivery.

#### 2.4 TRANSFER AND TURNOVER

Transfer and turnover agreements are important to software maintenance activities because they define responsibilities and methods for controlling error corrections and changes during the Deployment Phase. The CRISP provides an ongoing plan leading to the Program Management Responsibility Transfer (PMRT) and system/equipment turnover for C<sup>3</sup> systems. The PMRT transfers responsibility for engineering support to the supporting command (see AFR 800-4) and the turnover agreement (see AFR 800-19) transitions the operational system to the using command for operational use. The Computer Resource Working Group (CRWG), which is composed of representatives of all three commands, should assure that agreements incorporated in the CRISP are in the system turnover and transfer agreements. The contents of the PMRT and turnover agreements for computer resources are summarized in Chapter 9, Volume II, of AFR 800-14.

#### 2.4.1 Configuration Management

After system/equipment turnover and PMRT, the supporting command (normally an Air Logistics Center) is the system configuration change control authority. The supporting command while retaining engineering responsibilities may delegate to the using command control over those computer programs required for the direct performance of the operational mission (see AFR 102-5, Section A, 6, for special provisions for Command and Control Systems). In this situation, the using command will establish a Computer Program Configuration Sub-Board (CPCSB) to facilitate computer program change processing. The responsibilities of the CPCSB should be outlined in the CRISP and detailed in the Operational/Support Configuration Management Procedures (O/SCMP).

The implementing and supporting commands should use the VDD and a Specification Change Notice (SCN) to describe and distribute the program changes to a base-lined CPCI. The VDD and SCN are defined in MIL-STD-483, Appendix VIII. When the using command has configuration management responsibilities for a computer program the VDD and SCN, or other methods described in the O/SCMP, may be used to distribute CPCI changes.

The O/SCMP details how the basic configuration management approach, defined in the CRISP, will be implemented. Configuration management procedures should be written by the supporting and using commands. These procedures will be used during the Deployment Phase, but must be written during the Full Scale Development Phase. At a minimum, they should address the following items:

- The relationships of all commands involved
- The method for processing changes
- Approval authority for changes
- The status accounting procedures and responsibilities
- Handling of emergency changes
- The method for distributing CPCI changes and documentation
- Situations where turnover precedes PMRT

#### 2.4.2 Change Processing During Transfer and Turnover

Between transfer and turnover, and while the implementing command is developing the update changes identified in the PMRT agreement document, the implementing, using, and supporting commands may have a change control and coordination problem.

During this period:

- The implementing command must schedule time for fixes.
- The DT&E tested version of the proper software baseline is required (which itself may be undergoing modification).
- All affected documentation must be updated.

These activities require close coordination between the implementation, using, and supporting commands. A version release of all changes and modifications will facilitate change processing during transfer and turnover. The using and supporting commands should minimize changes until the implementing command has completed all of its scheduled update changes. The procedures for handling change processing during transfer and turnover should be spelled out in the PMRT agreement.

#### 2.4.3 Software Documentation

All operations and support documentation needed to operate, modify, maintain and otherwise support the system after PMRT should be identified in the CRISP and included in the CDRL. These documents should be approved at PCA, prior to turnover of the system. The uses of the documentation, including formats, should be discussed in the CRISP and O/SCMP.

#### 2.5 MAINTENANCE DURING DEPLOYMENT PHASE

The Operations and Maintenance (O&M) or Deployment Phase follows the Production Phase and PMRT. As discussed previously, the CRISP forms the foundation for the transfer and support of the embedded software in major systems. The CRISP is initiated in the Validation Phase to define computer resource requirements and is updated during the Full-Scale Development Phase. The CRISP incorporates using and supporting command requirements. It is the primary vehicle for identifying responsibilities for technical and management support of the operational software and computer hardware and related support tools and facilities. It establishes planning for the configuration management procedures to be followed by the using and supporting commands.

Before PMRT, the implementing, using, and supporting commands should have resolved the following transfer questions:

- What software will the using and supporting commands control respectively?
- If it is a multiple-site system, where will software be maintained and how will changes get to the other sites? A general solution to the multiple-site problem is to have a single overhead facility produce and validate changes and then ship new tapes with site-unique adaptation. This approach improves configuration control, allows knowledgeable programmers to make changes, and minimizes the programming staff needed at the other sites, all of which provides for a more efficient and cost-effective operation.
- Who is responsible for programmer training, documentation, maintenance facility operations, upgrades to new operating system (OS) releases, and to the PSL? The answer to these questions should be based on the maintenance concept.
- How will the using command's configuration management procedures interface with those of the supporting command?

If modification is too large or complex for the responsible command's maintenance staff, or if it involves hardware, the procedures of AFR 57-4 will be followed. Further, software documentation should not be maintained under the Technical Order system, but can be maintained as system-unique manuals. Procedures should be established between the supporting and using commands for releasing new versions of a program or corrections to programs.

During the Deployment Phase, the responsible programming agency will establish a maintenance organization and provide configuration management and software modification procedures that satisfy mission requirements. It is important that the maintainable attributes of the software be retained throughout the system life cycle.



### SECTION 3 - APPLICABLE REGULATIONS, SPECIFICATIONS, AND STANDARDS

This section discusses those directives, regulations, specifications, and standards (RSSs) that impact software maintenance. In general, the RSSs refer to operational support and modification, whereas this guidebook refers to these activities as software maintenance. Although little is said in the RSSs about software maintenance, it is specifically addressed in DoD Directive 5000.29 and AFR 800-14, Volume I. As used in this guidebook, the definition of software maintenance includes the ability to modify the software; therefore, the regulations covering configuration management are included in this discussion.

#### 3.1 RSSs WITH DIRECT IMPACT ON SOFTWARE MAINTENANCE

The following RSSs directly reference software maintenance and the allocation of maintenance responsibilities between the using and supporting commands:

- Department of Defense Directive 5000.29, "Management of Computer Resources in Major Defense Systems." This directive establishes DoD policy for the management and control of computer resources during system acquisition. Maintainability of both software and hardware is called out as a major consideration during initial design. In addition, DoD 5000.29 directs that support items required for cost effective maintenance be specified as deliverable items. It also requires the use of HOLs. Further, DoD 5000.29 establishes software maintainability as one of the prime items to be considered during system acquisition and directs all DoD components to develop and implement a disciplined management approach to providing effective software at minimum life cycle cost.
- Department of Defense Instruction 5000.31 "Interim List of DoD Approved Higher Order Programming Languages (HOL)." Specifies the HOLs which are approved for use in conjunction with DoDD 5000.29. Although this instruction allows for certain exceptions, it attempts to reduce proliferation and ensure control of HOLs in defense systems by limiting new development to six approved languages: CMS-2, SPL-1, TACPOL, JOVIAL, COBOL, and FORTRAN.
- Air Force Regulation 300-10, "Computer Programming Languages." Implements DODI 5000.31. This regulation restricts approved languages to FORTRAN, COBOL, JOVIAL (J3), and JOVIAL (J73/I), but adds PL/I. PL/I is not approved by AFSC for 800-series acquisitions, however.
- Air Force Regulation 800-4, "Transfer of Program Management Responsibility." States Air Force policy and assigns responsibility for the transfer of program management responsibility from an implementing to a supporting command.

- Air Force Regulation 800-14, "Management of Computer Resources in Systems." This regulation is presented in two volumes as follows:
  - Volume I establishes Air Force policy for the acquisition and support of computer equipment and computer programs that are dedicated elements of embedded systems. It establishes responsibility for maintenance and modification of computer programs; requires that organizational responsibility and computer resource requirements be established early in the acquisition cycle (including documentation, training, personnel, support facilities, and other essential resources); assigns specific management items (associated with computer resources) to be included in the PMD and PMP; and assigns responsibilities to both the supporting and using commands for acquiring facilities to support the maintenance, modification, and development of computer programs.
  - Volume II provides guidance for the planning and acquisition of computer resources, including support software and hardware. It establishes procedures for implementing the policies outlined in Volume I. This volume includes a definition of the phases of the system acquisition life cycle with special attention given to the computer program development process. Individual chapters are devoted to planning, engineering management, testing, configuration management, documentation, identifying contractual requirements, turnover and transfer, and support. The chapter on planning identifies the major planning documents associated with computer resources as follows:
    - Program Management Directive (PMD)
    - Program Management Plan (PMP)
    - Computer Resources Integrated Support Plan (CRISP)
    - Computer Program Development Plan (CPDP)
  
- Air Force Regulation 800-19, "System or Equipment Turnover." Establishes policy and principles for the efficient turnover to an operating command of systems or equipments developed under the program management concept established in AFR 800-2. Attachment 1 thereto calls for the timely identification of post turnover maintenance requirements and planning for their adequate implementation. Specific types of requirements cited include:
  - Manpower
  - Support and training equipment
  - Spares
  - Documentation
  - Facilities
  - Budgets
  - Information
  - Contractor Services
  - Computer resource requirements

In addition to the above documents, the following regulations covering configuration management are pertinent to the maintenance of software:

- AFR 57-4, "Retrofit Configuration Changes." This regulation establishes policy and provides guidance for obtaining approval for modifications to configuration items after the item has been put into service by an Air Force agency. It includes criteria for determining the classification of a proposed change and discusses the approval authority and procedures to be followed in submitting a change request for approval.
- AFR 65-3, "Configuration Management." This regulation establishes uniform policy and guidance to all DoD components in implementing configuration management procedures for all configuration items. It has chapters devoted specifically to configuration identification, control, status accounting, and audits. Appendix F contains Air Force implementation instructions which outline specific Air Force policy and assign responsibilities to various Air Force elements.
- AFSCM/AFLCM 375-7, "Configuration Management for Systems, Equipment, Munitions, and Computer Programs." This manual is the AFSC implementation of AFR 65-3.

### 3.2 IMPLEMENTATION OF RSSs

The regulations and directives identified in 3.1 require that the PO acquire software that is supportable. For software, ease of modification is as important as ease of correction (or repair). This characteristic must be built into the software throughout the acquisition cycle. It includes the design and coding of the computer program as well as support items, such as documentation, training, support software, and facilities.

Major considerations incorporated into the regulations include: the requirement for acquiring software that supports correction, modification, and growth; the inclusion of the using and supporting commands in all phases of acquisition planning; the concept of tailoring the management techniques to the specific system; and the requirement that support equipment and software be acquired as a part of the system.

The policy outlined in these regulations calls for an orderly development of systems through a series of plans, specifications, and baselines which are supported by appropriate engineering studies and tradeoffs.

The initial planning for software maintenance should begin with the development of initial system requirements. As a part of requirements analysis, a computer program support concept should be developed which considers the system mission, number of installations, the operational availability requirements, and expected level of change activity. This support concept should be

developed by the implementing command in consultation with the using and supporting commands and is essential to the system engineering efforts. AFR-800-14, Volume II, Chapter 3, provides special guidance concerning items to be considered.

The following planning documents must also address operational support and modifications.

- Program Management Directive. The PMD should address the support requirements needed to achieve mission objectives. If not included in the PMD, the implementing command should address these subjects in the PMP.
- Program Management Plan. Although the PMP has no specific sections addressing operational support and modifications (see AFSCP 800-3, Attachments 3 and 4), it requires detailed analysis of these items to support its sections on: Program Management (Section 3), System Engineering and Configuration Management (Section 4), Test and Evaluation (Section 5), Operations (Section 7), Manpower and Organization (section 10), and Personnel (Section 11).
- Computer Resources Integrated Support Plan. The CRISP identifies responsibilities and resources required for software maintenance after transfer and turnover and forms the basic agreement between the using and supporting commands. A comprehensive CRISP provides the basis for the smooth transfer and turnover of a system which has the support facilities and tools necessary for cost effective maintenance.
- Computer Program Development Plan. The CPDP\* is specified as a Contract Deliverable Requirements List (CDRL) item which is prepared by the contractor and approved by the PO. This plan gives the PO visibility into the software development plan and management necessary to procure adequate support software. The plan should be developed or reviewed with the objective of providing software which most effectively accommodates the computer program support concepts developed in the CRISP. This objective should include the structure of the software, documentation, and support tools. It should also address standards and conventions which can be used to enhance maintainability.

These four documents form the planning foundation necessary to acquire a software maintenance capability but they must in all cases be followed by a set of baselined contract specifications.

\*Defined in AFR 800-14, Volume II, Chapter 3 [see DI-S-30567].

### 3.3 POTENTIAL PITFALLS IN APPLYING RSSs

The RSSs do not provide adequate guidance for acquiring maintainable software. They do address support and modification and these terms have been included in the definition of software maintainability presented in Section 1.

The SD must recognize that planning for software maintenance is usually not given adequate attention by the using command when attempting to get an operational requirement validated. With long development cycles for large systems, software maintenance tends not to be an immediate concern and seems relatively minor to the user. However, if the PO does not insist on a concept for software maintenance it is almost impossible for the SD to develop the requirements for proper support.

Maintainable software and support capabilities cost money and must be budgeted, planned, and specified in the contractual specifications. Therefore, the PO must specify in the contractual specifications those software design and construction techniques which enhance software maintainability. In most cases, these same techniques also enhance quality, management visibility, and scheduling confidence. However, design and construction techniques specified in contractual specifications may impose design constraints that may not allow the contractor to meet all performance requirements. These constraints and support items may also result in increased total acquisition costs. Tradeoffs concerning the most important requirements with regard to the mission and total life cycle costs must therefore be considered by the PO.

The SD must exercise care, when specifying a particular hardware configuration, or when supplying it as GFE, that the hardware does not place undue constraints (affecting software maintainability) upon the contractor. If, for example, a lack of computer capacity forces the contractor into complex software design, the maintainability characteristics of the software will assuredly be jeopardized. The same care must be taken in specifying GFE software. If the characteristics or capabilities of the GFE software force the contractor into unduly complex interfaces or design, maintainability will suffer.

In addition to the acquisition management requirements for documentation, the PO should tailor documentation requirements to the software maintenance concept. The flow diagram requirements of MIL-STD-483 may be too detailed and unnecessarily costly for specific programs. For example, a recently acquired system required detailed flow diagrams which were prepared at great expense and then discarded immediately following turnover because the user didn't need them. The format and content of all documentation should be designed to fit the using and supporting commands' requirements and methods of updating. A major problem for these commands is the maintenance of software documentation after turnover. This can be facilitated by tailoring standard DIDs in the Full-Scale Development Phase RFP.

The PO should always assume that requirements for C<sup>3</sup> systems will change during the acquisition cycle. Since the acquisition cycle tends to be long, new requirements will assuredly be identified and, in many cases, original requirements will be deleted. For this reason alone, the PO should recognize that software modifiability is a desirable characteristic even though the anticipated level of change after turnover is low. It is therefore beneficial to the PO to stress software maintainability for its modifiability aspects alone. It is also cost effective from the standpoint of ease of error correction during System DT&E.

## APPENDIX A - DESIGNING MAINTAINABLE SOFTWARE

### 1. INTRODUCTION

This appendix discusses the design and development of maintainable software. The information provided should be used by the PO when reviewing the design philosophy documented in the contractor's proposal and CPDP. It should also be used to prepare for design reviews (PDR and CDR) when the contractor presents his overall and detailed design. Finally, it should be referenced when evaluating contractor compliance with his plans and design at PCA.

### 2. THE PROPERTIES OF MAINTAINABLE SOFTWARE

Software can be designed to implement either a single specification or a class of programs which respond to the specification. In general, the broader the design, the easier it is to change, but the more it will cost to design and operate. The software designer balances flexible design against ease of implementation or speed of execution. The designer looks for simple computational structures which will generate the computer program specified and yet be easily modifiable.

No techniques are known for optimizing design. However, a good flexible design has properties which can be evaluated by experienced designers at design reviews, and measured after implementation by an audit of maintenance costs.

Flexible design includes a coherent conceptual organization of data and operations, cleanly modularized design with low levels of interconnection, self-monitoring properties, hooks for easy extension, and a reasonable methodology to insure a complete and correct design. The following paragraphs define the properties of flexible design in more detail. Some criteria such as coherent conceptual organization and a closely modularized design are closely related. However, these properties differ in intent and even though they tend to come together it is possible to have one without another. These properties can be evaluated by a technically competent design critic in the same sense that good writing style can be evaluated by a good literary critic. Unfortunately, it is hard to find a good serious critic.

## 2.1 Coherent Conceptual Organization

A software design should be developed from a consistent set of design principles because once these principles are understood, the design becomes predictable and hence easier to modify.

A coherent conceptual organization allows the prediction of the information that will be found in data structures and the functions that will operate on this data. If there are communication conventions between modules then one would expect to know what the conventions are and why they were chosen. In general the design philosophy should tell a reader the problems perceived by the designer and how they were solved.

When a software design for a large system is being developed, several levels of abstraction should be defined. Each level of abstraction should be defined in terms of data types and operations on these types. This technique forms a coherent approach for the total system by collecting related functions at appropriate levels of abstraction.

Software must be built with consistent philosophy and organization as well as a clear understanding of the design problems to be solved. Without such an approach, maintenance becomes much more difficult because there are no principles to follow in determining how modifications were made. Whenever a coherent design philosophy is not developed or successfully communicated to the maintenance programmers, modifications become more expensive and are likely to contain residual errors.

## 2.2 Modular Design

The goal of modular design is to build independent functional pieces of a computer program which can be separately developed, tested, and modified or replaced. Good modular design minimizes coupling between subroutines and procedures and allows easy modification. Software which employs modules must rely only on a limited, well defined set of properties for the modules. Nothing in the software should depend on the internal method by which the modules accomplish their job. Proper modularity will reduce code by collecting similar functions into one module. It increases the program's clarity by employing a small conceptual set of well defined properties to construct larger program elements. Figure 6 lists design approaches which minimize the coupling of modules and Figure 7 provides design approaches which assist modularization.



METHOD	IMPACT ON COUPLING
Minimize the sharing of a common environment; i.e., common data files and tables.	Two or more modules sharing a common data environment increase interface complexity and coupling. Every element in the common environment, whether used by a particular module or not, constitutes a separate path along which errors and changes can propagate. Once the choice is made to communicate via a common environment, all new modules must be plugged into the common environment, further compounding the total complexity. These disadvantages can be minimized by limiting access to the smallest possible subset of modules (subset data into groups).
Reduce interface complexity.	Reducing complexity of an interface reduces the information needed to state or understand the connection. Thus, obvious relationships result in lower coupling than obscure or inferred ones.
Avoid referring to the contents of a module.	Connections that address or refer to a module by its name, rather than its contents, yield lower couplings than connections referring to a module's internal elements. Modules that can be used without knowledge of their contents make for simpler systems.
Minimize control information passed between modules.	By avoiding the practice of passing an "element of control" such as a switch, flag, or signal from one module to another, coupling is reduced. Passing an "element of control" affects the execution of another module and not merely the data with which it works by involving one module in the internal processing of another module.
Maximize module cohesiveness.	Coupling is reduced when the relationships among elements not in the same module are minimized. There are two ways of achieving this--(1) minimizing the relationship between modules and (2) maximizing relationships between elements in the same module. An "element" is any form of a "piece" of the module, such as a statement, a segment, or a subfunction. Binding is the measure of cohesiveness of a module. The objective is to reduce coupling by striving for high binding. Functional binding is the strongest type of binding. In a functionally-bound module, all elements are related to the performance of a single function. Examples of functionally bound modules are "Compute Square Root," "Obtain Random Number," "Write Record to Output File."

Figure 6. Design Approaches to Increase Module Independence.

APPROACH	DISCUSSION
Match Program to the class of problem being solved.	One of the most useful techniques for reducing the effect of changes on the program is to make the structure of the design match the structure of the class of problem being solved, i.e., form should follow function.
Keep the scope of the effect of a decision within the scope of the control of the module effected.	The scope of control of a module is that module plus all modules that are ultimately subordinate to that module. The scope of effect of a decision is the set of all modules that contain some code whose execution is based upon the outcome of the decision. The system is simpler when the scope of effect of a decision is in the scope of control of the module containing the decision. The scope of effect can be brought within the scope of control either by moving the decision element up in the structure or by taking those modules that are in the scope of effect and moving them so that they fall within the scope of control.
Limit size of module.	There is no consensus on an optimum module size. IBM, in their paper on "Chief Programmer Team Management of Production Programming," suggests that programmers write modules of approximately 50 PL/I statements. This can be kept on a single page and is readily comprehensible. Others have suggested from 100-300 programming statements. In any case, the intent is not to set absolute maximums but to guide programmers, through the application of a size standard, to functionally orient their code to improve readability and to provide modularity.
Other considerations	<p>Eliminate duplicate functions but not duplicate code. When a function changes, it is advantageous to only have to change it in one place. This does not mean that the module can not be used in more than one place in the system, (e.g., the use of MACROS should be encouraged). If a requirement changes in one part of the system, then a new module should be developed with a different name to maintain independence. An example might be a match function used by multiple CPCs. If one CPC needed more accuracy, a separate routine would be developed to prevent impacting other CPCs.</p> <p>Check Modules that have many callers or that call many other modules. While not always a problem, it may indicate missing levels of modules (e.g., an erroneous functional boundary of that module).</p> <p>Isolate all independencies on a particular data-type, record-layout, index-structure, etc., in one or a minimum number of modules. This minimizes recoding should the specification change.</p> <p>Reduce the number of parameters passed between modules. Do not pass whole records from module to module. Pass only the field or fields necessary for each module to accomplish its function, otherwise, all modules will have to change if one field expands, rather than only those which directly use the field. Passing only the data being processed by the module with necessary error and End of File (EOF) parameters is the ultimate objective.</p>

Figure 7. Some Approaches for Defining Modules.

### 2.3 Self-Monitoring Computer Programs

Active and passive monitoring are two forms of self-monitoring which are valuable for maintenance and should be considered for inclusion into the design.

Active monitoring goes on continuously and is designed to verify the reliability of computer program data. Techniques from financial auditing can sometimes be adapted to perform this kind of checking. For example, if the records in a file are sequentially numbered, a missing record can be detected. A checksum of items included as the last data item can be used to detect unwanted changes to the data. Often this kind of checking can occur with almost zero operating time costs by making the checks a by-product of normal processing.

Many of these active monitoring techniques will increase the reliability of a computer program which is being modified by detecting changes to fixed assumptions being used by the computer program code. Modifying and extending programs which include this form of checking is always easier and more secure because if an error occurs in one module it is likely to be detected by other modules in the same program, thus warning maintenance personnel of an error.

Passive monitoring is activated by maintenance personnel and allows information to be optionally checked or initiates the processing of trace information to assist in evaluating program operation after maintenance activity has begun. Passive monitoring provides diagnostic assistance in locating a module which is not performing properly and assists in the integration of new modules. Software designed for flexibility should include support modules which can provide the programmer with visual representations of the data structures and detailed analyses of the computer program's actions.

### 2.4 Program Hooks for Future Extensions

Software which is designed for future extension will usually include fields in data structures that can be used for future modifications to the software without changing existing programs which use the structures. Such fields are called hooks and save valuable maintenance time when the original software is extended. In choosing appropriate hooks, it is prudent to spend some design time thinking about how a new feature might be integrated into the existing software design.

### 2.5 Design Methodology

A design methodology should be chosen to insure the completeness and quality of the resulting software design. Most good computer program design methodologies begin with functional requirements, which are stated in the Development (Part I) Specification, and progress downward at progressively greater levels of detail until code is produced. This technique is generally referred to as top-down design.

Top-down design is accomplished by successively refining a computer program description to meet Development Specification requirements. Each time a refinement is accomplished the substructure inherits an allocated set of performance requirements. If each substructure can be constructed within the Development Specification then the whole CPCI will meet its performance criteria.

Strict top-down design is difficult unless the entire CPCI is constructed with a similar organization. Otherwise it is hard to establish the appropriate performance criteria for substructural elements or even insure that some divisions can be accomplished at all. Top-down design is illustrated in Figure 8 where successive levels of design provide additional details of the eventual solution.

To be effective, a design methodology must be easily communicated and understood. In developing a design, it is useful to examine the system from the following four points of view:

- System Physical Structure. Review all the system components and their relationships. A description of the physical structure of the system is usually contained in the System Specification. More detailed descriptions of the CPCI and all its interfaces should be included in the Development Specification (see Computer Program Development Specification guidebook).
- Functional Decomposition of the CPCI. Analyse the decomposition to determine the hierarchy of control and what is to be done by the CPCI. Normally, time is not represented in a functional decomposition. Figure 8 shows an example of a top-down functional decomposition.
- CPCI Data Flow. Examine data flow to identify all inputs and outputs and ascertain the flow of information as it proceeds through the functional areas of the CPCI.
- Dynamic Operations of the CPCI. Determine the required sequential and concurrent operations and identify processing volumes and priorities to provide a basis for evaluating the throughput and response time features of the design.

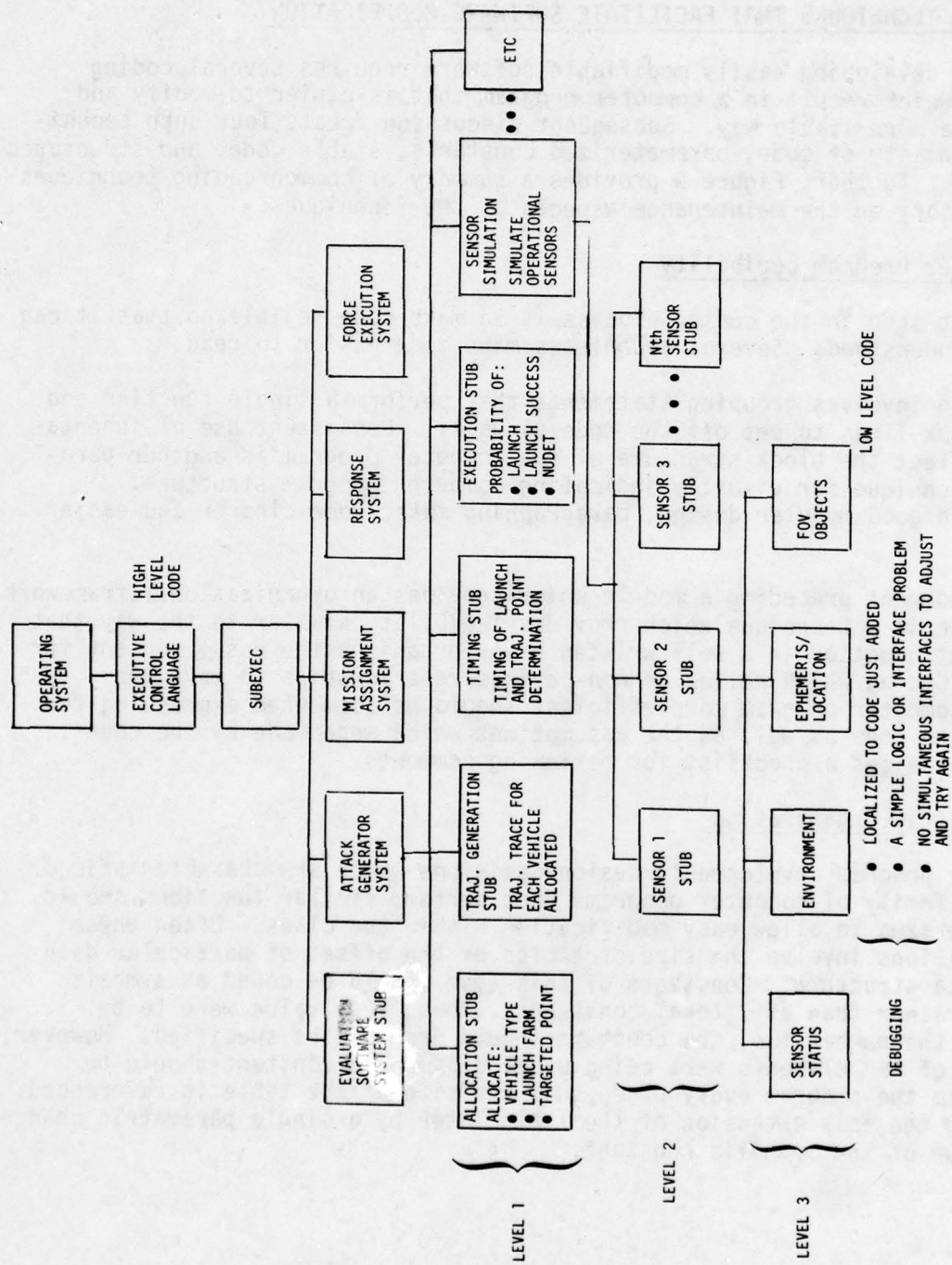


Figure 8. Top Down Development

### 3. CODING TECHNIQUES THAT FACILITATE SOFTWARE MODIFICATION

The task of developing easily modifiable software requires several coding techniques which result in a computer program that is easier to modify and behaves in a more stable way. Subsequent discussion treats four such techniques: legibility of code, parameterized constants, stable code, and structured programming. Further, Figure 9 provides a summary of common coding techniques with commentary on the maintenance aspects of the techniques.

#### 3.1 Computer Program Legibility

An important step in the coding process is to make code legible so that it can be easily understood. Several techniques make code easier to read.

Paragraphing involves grouping statements that perform a single function and placing blank lines to set off the code visually. Consistent use of indentation to reflect the block structure of the computer program is another paragraphing technique for visually indicating computer program structure. Coupled with good modular design, paragraphing makes code clearer and easier to modify.

A general comment preceding a module which provides an organizational framework for the code is a technique which provides legibility similar to the way that a proper introduction in a well written essay organizes the essay content for a reader. Coding which relies on non-obvious relationships to make the resulting computer program more efficient should be commented expressing the intent of the code as well as the assumptions which were made by the coder. Figure 10 provides a checklist for reviewing comments.

#### 3.2 Parametric Organization

In computer program development, design decisions which are characteristic of a class or family of computer programs that perform similar functions should be parameterized to allow easy modification within the class. Often these design decisions involve the size of tables or the offset of particular data in some data structure. Constants of this type should be coded as symbolic constants rather than as literal constants. Thus, if a value were to be divided by the number two, the constant should probably be specified. However, if a table of ten elements were being used, a symbolic constant should be written into the program every place that the size of the table is referenced. This allows the easy extension of the table later by a single parametric change to the value of the symbolic constant.

TECHNIQUES	EXAMPLE/COMMENTS	DISADVANTAGES
Code in a single HOL when possible.	A requirement of DoD Directive No. 5000.29, "DoD Approved Higher Order Programming Languages (HOLs), states that HOLs will be used to develop Defense system software, unless it is demonstrated that none of the approved HOLs are cost effective or technically practical over the system life cycle." DoDD 5000.31 lists the approved HOLs.	HOL programs usually take longer to operate and require more core storage than assembly language. Often time-critical portions of real time systems are coded in assembly language.  The maintainability of using one HOL may be outweighed by the benefit of using several off-the-shelf packages.
Use symbolic parameters to represent constants, relative location within a table, and size of data structures.	Accommodates changes in constants, table structure, sizes of tables, etc., without major changes to module code. For example, all software system constants can be centrally located and symbolically defined (TWOPI=2).	Unless care is taken, can confuse symbolic constants with variables in the program.
Sharing variables and temporary storage.	Each module (subroutine) should have own temporary storage area. Coding problems resulting from shared storage are hard to isolate since they can be coupled through a timing or interrupt relationship.	Could require additional core storage or additional processing time to housekeep temporary storage.
Subroutine arguments instead of global common to communication data.	Cuts down number of modules affected if global common is changed. By passing explicit parameters, module independence is enhanced.	Requires extra time to operate module.
Self modifying code.	Besides being hard to debug and modify, this type of code is extremely hard to follow.	Advantages far outweigh any potential core, or execution time savings.
Named COMMON instead of blank COMMON.	Items in COMMON should be named to allow referencing (not COMMON + x words). If referenced by name, changes to COMMON will have less impact on modules.	None.
Code which implicitly couples one module to another.	Destroys module independence and is hard to modify. Enforce the concept of one entry and one exit point.	Could require additional core.
To the extent possible minimize hardware dependencies.	For example, (1) use an internal character set and convert incoming data to this set; (2) use I/O modules that are separate from computational modules to read input data; (3) use a HOL so that machine-peculiar features are not used directly by the programmers, etc.	Could require extra processing time and additional core storage.
Structured code.	Structured programming encourages straightforward control logic. If a structured language compiler is not available, a pre-processor can be developed fairly economically or proper coding conventions should be enforced.	Could require development of pre-processor and additional core storage. Pre-processor has inherent disadvantages in terms of portability and maintenance.
Avoid unnecessarily complicated arithmetic statements.	Simple, logical coding is easier to understand, correct, or modify.	None.

Figure 9. Summary of Coding Techniques.

Program code should contain sufficient information to determine or verify the code's objectives, assumptions, constraints, inputs, outputs, components, and revision status. Comments should meet the following checklist:

- a. Does each computer program module contain a header block of comments which describe:
  1. Computer Program name?
  2. Effective date (last revision)?
  3. Accuracy requirements?
  4. Purpose?
  5. Limitations and restrictions?
  6. Modification history (a list of changes added)?
  7. Inputs and outputs?
  8. Assumptions?
  9. Error recovery types and procedures for all foreseeable error exits that exist?
- b. Are decision points and subsequent branching alternatives adequately commented and is proper indentation used to show the block structure of the coding logic?
- c. Are the functions of the modules and inputs/outputs sufficiently described to facilitate module testing?
- d. Are comments provided to support selection of specific input values to permit performance of specialized program testing?
- e. Is information provided to support assessment of the impact of a change in other portions of the computer program?
- f. Do all computer program statements which have undergone modification (after baselining) have an identification number included that associates the change with an ECP or discrepancy report?
- g. Where there is inter-module communication, is it clearly specified by comments, computer program documentation, or inherent program structure?
- h. Are variable names descriptive of the physical or functional property represented?

Figure 10. Checklist for Commenting Computer Program Code



Symbolic constants should be identified by their function in the program and not by the particular value they currently possess. Thus "TEN" is not a very useful name for a symbolic constant whereas "TABLESIZE" is a much better name because it expresses the meaning within the computer program. One difficulty with symbolic constants is that they can be confused with variables when a program is being read. To compensate for this deficiency constants should be listed in a comment which defines their intended use.

In this way, properly used parametric constants make a program much easier to read and understand as well as easier to modify. A person reading the code does not have to guess what meaning is attributed to a parametric constant; however, a number is just a number and carries no extra information.

### 3.3 Stable Code

Stable Code performs predictably even when given improper data. Programs should check for improper data and bypass normal processing whenever improper data is encountered.

The Development (part I) Specification should include specific requirements for the processing of improper data inputs received from sources external to the CPCII. It should also provide for some type of error message when internally-generated, improper data is encountered. Internal CPCs may erroneously generate improper data and stable code must prevent the processing of the improper data. Without error messages, the software might ignore the improper data and the error could go undetected for some time.

### 3.4 Development Methodology

To develop understandable computer program code it is best to employ a good development methodology. A development coding methodology is like a style sheet: it forces acceptable expression but it cannot make a great writer out of a poor one.

The most familiar methodology is structured programming, a technique initiated by Dijkstra. The structured-programming, flow-of-control conventions are summarized in Figure 11. These rules attempt to make a program more linear and thus easier to read and understand. However complexity in data structures can usually be traded for complexity in flow-of-control within a program. If a programmer does not attempt to keep both data structures and flow-of-control simple the computer program will still be difficult to read and understand.

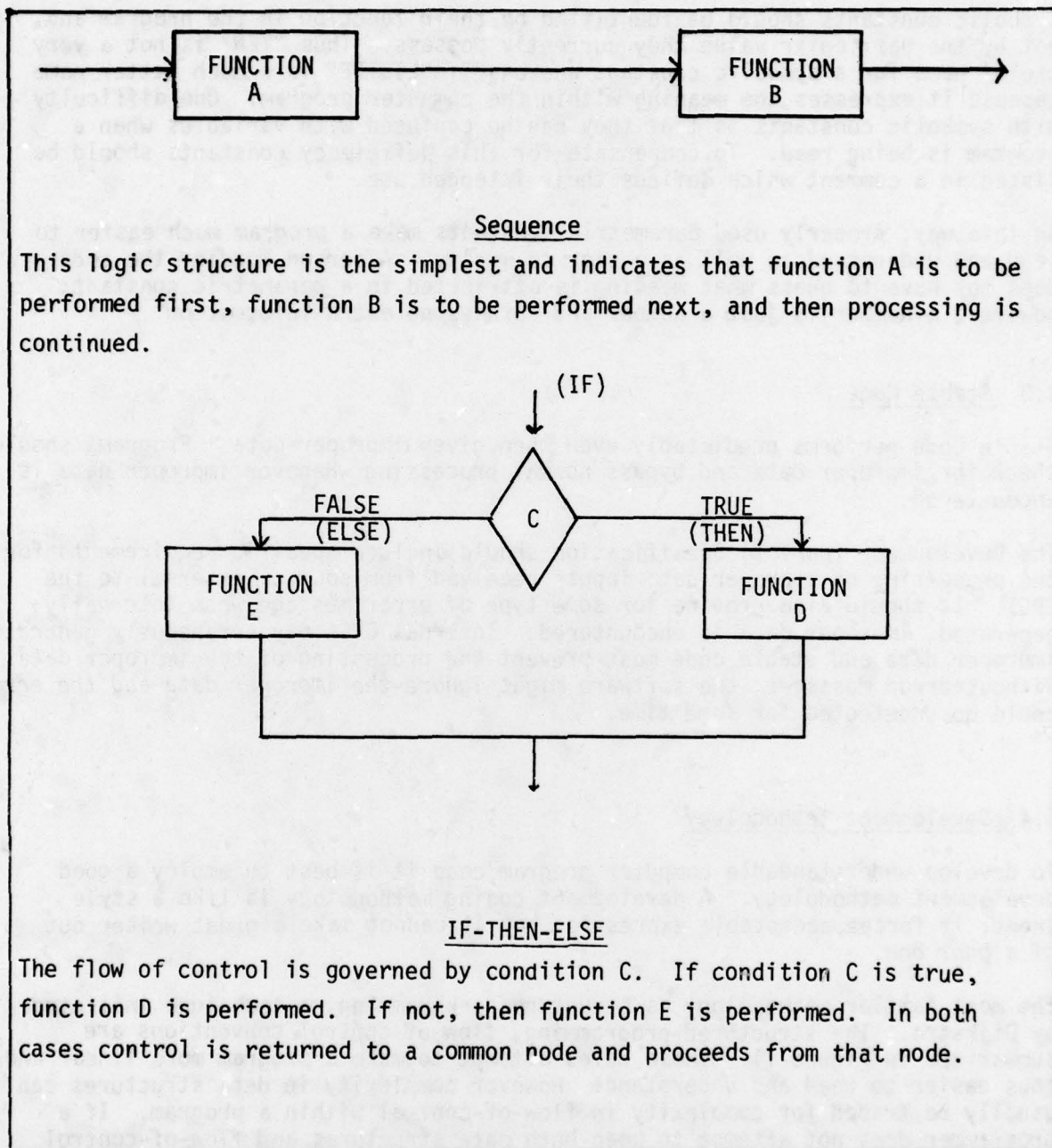


Figure 11. Structured Programming Basic Control Structures (1 of 3)

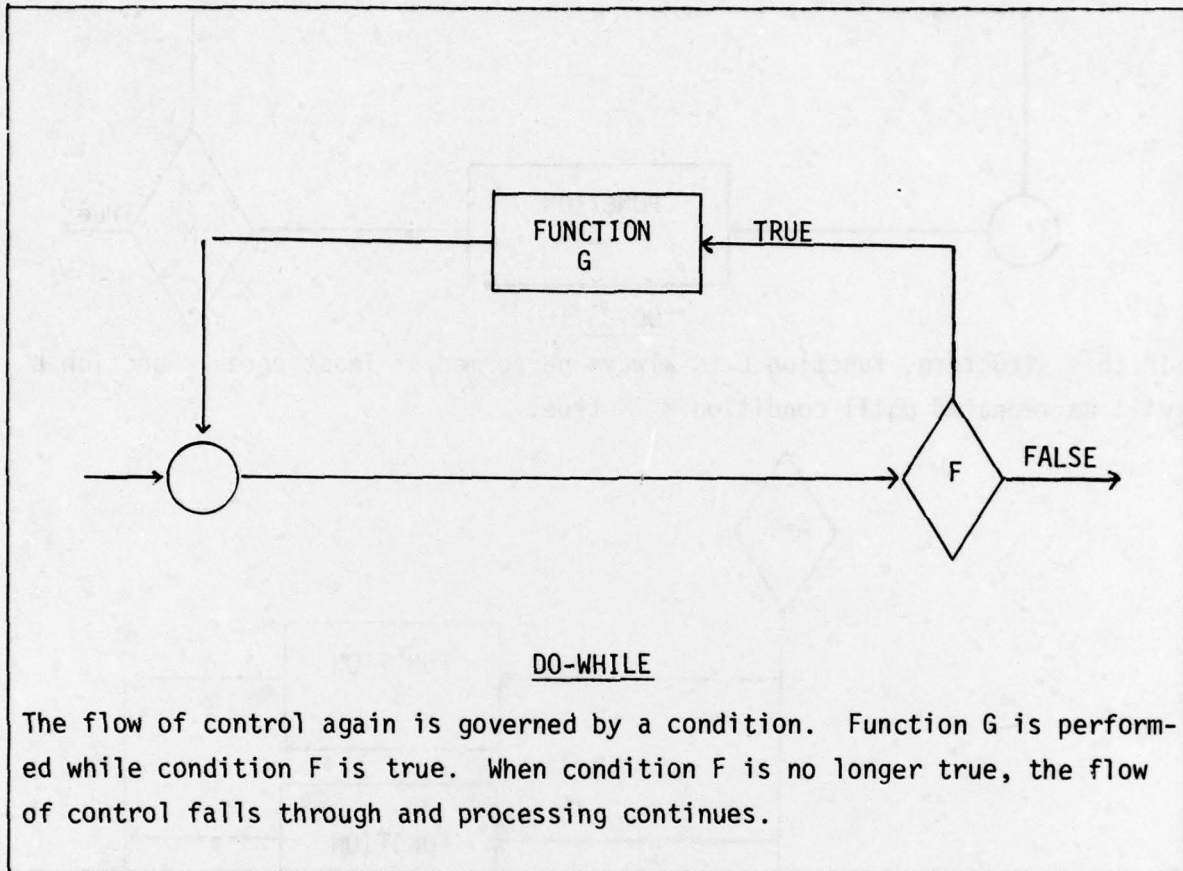
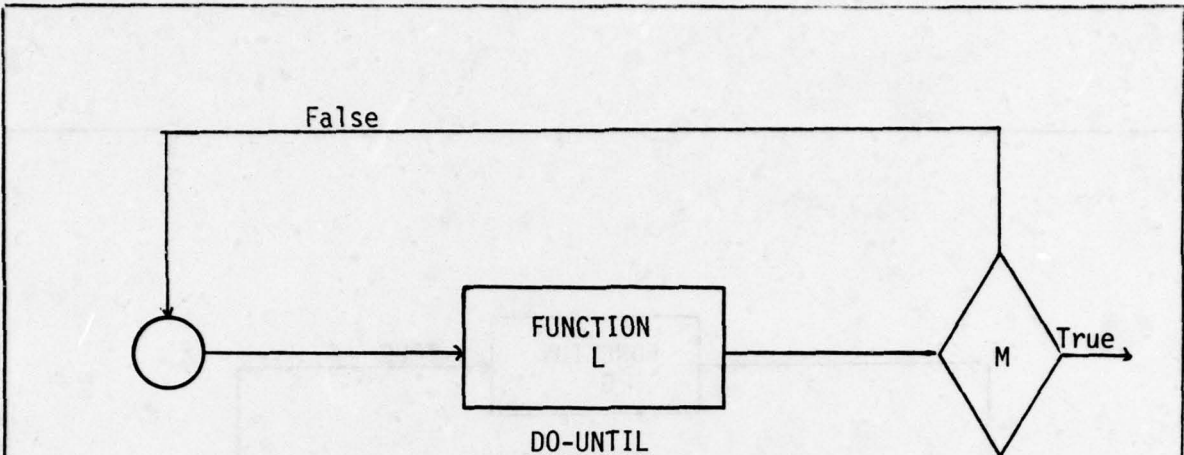
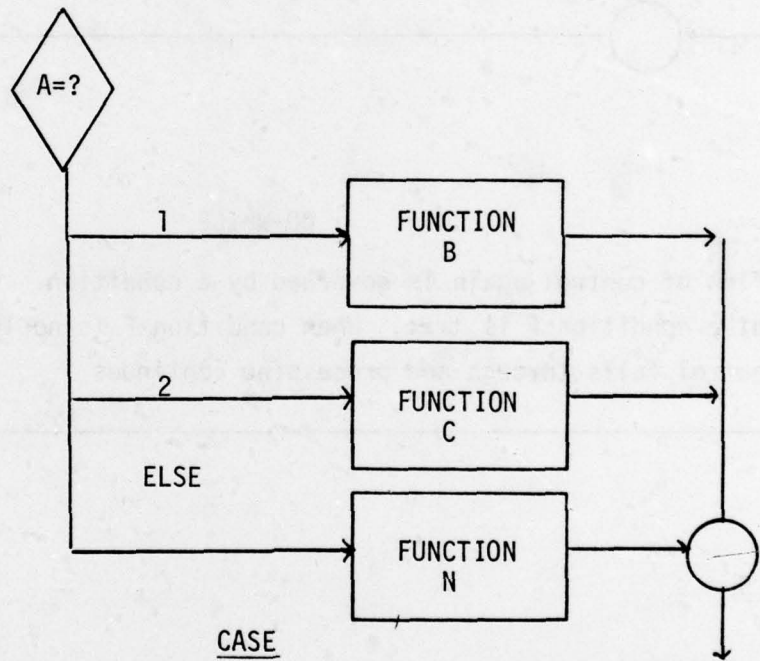


Figure 11. Structured Programming Basic Control Structures (2 of 3)



DO-UNTIL

In this structure, function L is always performed at least once. Function L will be repeated until condition M is true.



CASE

In the case logic structure, the flow of control is governed by the value of A. If A is 1 then function B is performed. If A is 2, C is performed. For all other values, a function N is performed. In all cases, control returns to a common node and proceeds from that node.

Figure 11. Structured Programming Control Structured Extensions (3 of 3)

## APPENDIX B - GLOSSARY

This guidebook consists of (1) definitions of major terms used throughout this guidebook and (2) a list of acronyms and abbreviations used herein.

### DEFINITIONS

Alter. A change to the source language representation of a program.

Assembly Language. A programming language which provides symbolic representations of machine-operation codes and addresses. In most cases, assembly language instructions generate computer object code on a one-to-one basis.

COMPOOL. A description of the components of a programming system. These components may be programs, common functions, and data shared by more than one component. The description exists in a symbolic and a machine representation. A COMPOOL is not a system data base but rather a description of its structure.

Computer Data. Basic elements of information used by computer equipment in responding to a computer program.

Computer Program. A series of instructions or statements in a form acceptable to computer equipment, 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/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 be designed to satisfy the requirements of a specialized process or particular users.

Computer Software. A combination of associated computer programs and computer data required to enable the computer equipment to perform computational or control functions.

Debug. A general term given to the process of finding and correcting the errors that have been shown to exist in a program by the testing or execution of a computer program.

Global Data. Global data are data shared by more than one module.

High Order Language (HOL). A machine-independent programming language in which the characteristics of a particular computer are not apparent.

HIPO (Hierarchical Input Output) Chart. The HIPO chart describes functions in terms of the inputs to a process, the process, and the outputs resulting from the process. The inputs, processes, and the outputs are arranged graphically on a page from left to right with directional arrows to lead the reader from input to output through the process.

Implementing Command. The command charged with primary responsibility for developing and acquiring the system or equipment.

Module. Used in this document to describe the smallest computer program unit that can be compiled or assembled. A CPC has one or more modules.

Object Code. Program code that results from the execution of a compiler or assembler.

Operating Command. The command or agency primarily responsible for the operational employment of a system, subsystem, or items of equipment.

Patching. Making changes to the machine-code representation of a computer program.

PMRT. See Transfer.

Program Support Library (PSL). A group of manual or automated procedures used to control and keep records of the developing software.

Source Code. Programmer-coded input to a program-language compiler or assembler (e.g., FORTRAN source statement).

Supporting Command. The command charged with primary responsibility for program management in the Deployment Phase including logistics, engineering, and procurements.

Top-Down Development. Top-down, also called stepwise refinement, is the name given to a methodology in which one starts at the level of the program to be solved and by a sequence of decompositions of the functional and data specifications finally arrives at the available machine or programming language.

Transfer. Refers to Program Management Responsibility Transfer (PMRT). The transfer of program management responsibility for a system (by series), or equipment (by designation), from the implementing command to the supporting command. PMRT includes transfer of engineering responsibility. (AFR 800-4)

Turnover. That point in time when the operating command formally accepts responsibility and accountability from the implementing command for the operation and organizational maintenance of the system or equipment acquired. (AFR 800-19)

## ACRONYMS AND ABBREVIATIONS

- AFR. Air Force Regulation
- AFSC. Air Force Systems Command
- C<sup>3</sup>. Command, Control, and Communications
- CDR. Critical Design Review
- CDRL. Contract Data Requirements List
- CPC. Computer Program Component
- CPCI. Computer Program Configuration Item
- CPCSB. Computer Program Configuration Sub-Board
- CPDP. Computer Program Development Plan
- CPU. Central Processing Unit
- CRISP. Computer Resources Integrated Support Plan
- CRWG. Computer Resource Working Group
- DID. Data Item Description
- DoD. Department of Defense
- DR. Discrepancy Report
- DT&E. Development Test and Evaluation
- ECP. Engineering Change Proposal
- ESD. Electronic Systems Division
- FCA. Functional Configuration Audit
- FQT. Formal Qualification Test
- GFE. Government Furnished Equipment
- HOL. High Order Language

I/O. Input/Output  
MIL-STD. Military Standard  
O&M. Operations and Maintenance  
OS. Operating System  
O/SCMP. Operational/Support Configuration Management Procedures  
PCA. Physical Configuration Audit  
PDR. Preliminary Design Review  
PMD. Program Management Directive  
PMP. Program Management Plan  
PMRT. Program Management Responsibility Transfer  
PO. Program Office  
PQT. Preliminary Qualification Test  
PSL. Program Support Library  
QA. Quality Assurance  
RADC. Rome Air Development Center  
RFP. Request for Proposal  
RSSs. Regulations, Specifications, and Standards  
SAM. Software Acquisition Management  
SCN. Specification Change Notice  
SD. Software Director  
SDR. System Design Review  
SOW. Statement of Work  
SRR. System Requirements Review  
TDSP. Top Down Structured Programming  
TR. Technical Report  
USAF. United States Air Force  
VDD. Version Description Document  
WBS. Work Breakdown Structure



### APPENDIX C - BIBLIOGRAPHY

"A Design Methodology for Reliable Software Systems;" Liskov, B. H.; Proceedings of Fall Joint Computer Conference; Vol. 41, Part 1; Pgs. 191-199; AFIPS; 1972.

"A Discipline of Programming;" Dijkstra, E. W.; Prentice Hall; Englewood Cliffs, N. J.; 1976.

"Designing Reliable Software;" Ogdin, J. L.; Datamation; Pgs. 71-78; July 1972.

"Modular Programs: Defining the Module;" Cohen, A.; Datamation; Vol. 18, No. 1; Pgs. 34-37; 1972.

"On the Criteria to be Used in Decomposing Systems into Modules;" Parnas, D. L.; Communications of the ACM; Vol. 15, No. 12; Pgs. 1053-1058; December 1972.

"Quantitative Analysis of Software Reliability;" Dickson, J. C., Hesse, J. L., Kientz, A. C., and Shoomun, M. L.; IEEE Symposium on Software Reliability; January 1972.

"Research Toward Ways of Improving Software Maintenance: RICASM Final Report;" Overton, R. K., et al; ESD-TR-73-125; USAF (ESD); January 1973.

"Reliability of Real-Time Systems;" Yourdon, E.; Modern Data; Serialized (six parts); January - June 1972.

"Scheduled Maintenance of Applications Software;" Lindhorst, W. M.; Datamation; Pgs. 64-67; May 1973.

"Software Development;" Mills, H. D.; Supplement to Proceedings of 2nd International Conference on Software Engineering; Pgs. 79-86; ACM/IEEE/National Bureau of Standards; IEEE Catalog No. 76CH1125-4C; October 1976.

"Software Engineering and Structured Programming;" Wilkes, M. V.; Supplement to Proceedings of 2nd International Conference on Software Engineering; Pgs. 132-134; ACM/IEEE/National Bureau of Standards; IEEE Catalog No. 76CH1125-4C; October 1976.

"Software Engineering;" Boehm, B. W.; IEEE Transaction on Computers; Vol. C-25, No. 12; Pgs. 1226-1241; December 1976.

"Structured Programming;" Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R.; Academic Press; London and New York; 1972.

"Structured Design;" Stevens, W. P., Meyers, G. J.; and Constantine, L. L.; IBM System Journal, No. 2; 1974.

"Structured Programming Series;" RADC-TR-74-300; Volume VI - Programming Support Library Program Specification; USAF.

"Techniques of Program Structure and Design;" Yourdon, E.; Prentice-Hall; Englewood Cliffs, N. J.; 1975.

"The Elements of Programming Style;" Kreitzberg, C. B., Shneiderman, B.; Modern Data; Pgs. 40-41; August 1972.

"The Influence of Software Structure on Reliability;" Parnas, D. L.; Proceedings of the International Conference on Reliable Software; April 1975.

"The Structure of 'The' Multiprogramming System;" Dijkstra, E. W.; Communications of the ACM, Vol. II, No. 5; Pgs. 341-346, 1968.

"Through the Central 'Multiprocessor' Avionics Enters the Computer Era;" William, A. O., O'Donnell, C.; Astronautics and Aeronautics; July 1970.

"Top-Down, Bottom-Up, and Structured Programming;" McClure, C. L.; IEEE Transactions on Software Engineering; Pgs. 397-403; December 1975.

"Top Down Programming in Large Systems;" Mills, H. D.; Debugging Techniques in Large Systems; Rustin, R. (Editor); Pgs. 41-55; Prentice Hall; Englewood Cliffs, N. J.

COMMENT SHEET

Software Maintenance Guidebook

Reviewer's Name:

Reviewer's Organization:

Comments:

Please return to: Hq ESD/MCIT (Stop 36)  
Hanscom AFB, MA 01731

(FOLD)

(FOLD)

FROM: \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

Hq ESD/MCIT

Stop 36

Hanscom AFB, MA 01731

64