

# SANDIA REPORT

SAND85-2346 • UC-32

Unlimited Release

Reprinted October 1992

## Sandia Software Guidelines Volume 3 Standards, Practices, and Conventions

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550  
for the United States Department of Energy  
under Contract DE-AC04-76DP00789



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
Office of Scientific and Technical Information  
PO Box 62  
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from  
National Technical Information Service  
US Department of Commerce  
5285 Port Royal Rd  
Springfield, VA 22161

NTIS price codes  
Printed copy: A06  
Microfiche copy: A01

Distribution  
Category UC-32

SAND85-2346  
Unlimited Release  
Printed July 1986  
Reprinted October 1992

# Sandia Software Guidelines

Volume 3

*Standards, Practices, and Conventions*

*Sandia National Laboratories  
Albuquerque, New Mexico 87185*

## Abstract

This volume is one in a series of *Sandia Software Guidelines* intended for use in producing quality software within Sandia National Laboratories. In consonance with the IEEE Standard for Software Quality Assurance Plans, this volume identifies software standards, conventions, and practices. These guidelines are the result of a collective effort within Sandia National Laboratories to define recommended deliverables and to document standards, practices, and conventions which will help ensure quality software.

## Foreword

This volume is one in a series of Sandia Software Guidelines intended for use in producing quality software within Sandia National Laboratories. These guidelines, when used in conjunction with the IEEE Standard for Software Quality Assurance Plans, will help ensure that computer programs developed within the Laboratories are usable, reliable, understandable, maintainable, and portable. When complete, the series will consist of the following documents:

- **Volume 1: Software Quality (SAND85-2344)**  
Presents an overview of procedures designed to ensure software quality. Includes a sample software quality assurance plan for a generic Sandia project.
- **Volume 2: Documentation (SAND85-2345)**  
Presents a description of documents needed for developing and maintaining software projects. Includes sample document outlines for a generic Sandia software project.
- **Volume 3: Standards, Practices, and Conventions (SAND85-2346)**  
Presents consensus standards and practices for developing and maintaining quality software at Sandia. Includes recommended deliverables for major phases of the software life cycle.
- **Volume 4: Configuration Management (SAND85-2347)**  
Presents a methodology for configuration management of Sandia software projects and their associated documentation.
- **Volume 5: Tools, Techniques, and Methodologies (SAND85-2348)**  
Presents evaluations and a directory of software tools and methodologies available to Sandia personnel.

## Acknowledgement

A consensus document like this volume of the guidelines cannot be produced without the cooperation and hard work of a great many people throughout the organization. The sponsoring CAD Technology Division wishes to thank the members of the working group who wrote Volume 3, as well as the members of the balloting group who reviewed and refined it.

## Working Group Preface

We have been proud to participate as members of the Working Group that has produced this document. We all believe that the ideas and practices documented herein are important and deserve your attention. Like all converts to a new way of thinking, our past and even present actions are not necessarily in line with the ideal. Few of us have had the opportunity to apply all of these practices to a complete project. We believe the practices we have personally used have helped to produce a higher quality software product.

### Working Group Members, Volume 3

Mike Blackledge, <i>Chairperson</i>		Mike McGlaun	(6444)
Doug Adams	(7262)	Darl Patrick	(7252)
Art Ahr	(2826)	Don Rountree	(5321)
Sandra Babb	(2854)	Suzanne Rountree	(2813)
Louann Grady	(2812)	John Wisniewski	(2113)
Dick Isler	(8274)	Ann Yates	(5255)

*This document was printed using the L<sup>A</sup>T<sub>E</sub>X computer typesetting program and the Sandia National Laboratories' Autologic APS 5 phototypesetter.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Intent . . . . .	1
1.2	Environment . . . . .	1
1.3	Applicability . . . . .	2
1.4	Organization . . . . .	3
1.5	How to Use This Manual . . . . .	3
1.6	Summary . . . . .	4
<b>2</b>	<b>Project Planning and Management</b>	<b>5</b>
2.1	Recommended Deliverable . . . . .	5
2.2	Why Project Planning? . . . . .	5
2.3	Project Plan . . . . .	6
2.4	Project Management . . . . .	7
2.5	Tools, Techniques, and Methodologies . . . . .	8
<b>3</b>	<b>Requirements</b>	<b>9</b>
3.1	Recommended Deliverables . . . . .	9
3.2	Why Requirements? . . . . .	9
3.3	Software Requirements Specification . . . . .	10
3.4	Software Requirements Review . . . . .	13
3.5	Prototypes . . . . .	13
3.6	Tools, Techniques, and Methodologies . . . . .	14
<b>4</b>	<b>Design</b>	<b>15</b>
4.1	Recommended Deliverables . . . . .	15
4.2	Why Design? . . . . .	16
4.3	Software Design Criteria . . . . .	17
4.4	General Design Standards and Guides . . . . .	18
4.5	Detailed Design Procedure . . . . .	19
4.6	Design Description Documents . . . . .	20
4.6.1	Preliminary Design Document . . . . .	20
4.6.2	Detailed Design Document . . . . .	21
4.7	Design Reviews . . . . .	22
4.7.1	Criteria for Design Reviews . . . . .	22
4.7.2	Preliminary Design Review . . . . .	22
4.7.3	Critical Design Review . . . . .	23
4.8	Methodologies and Tools . . . . .	23

4.8.1	Structured Design Techniques . . . . .	24
4.8.2	Software Design Tools . . . . .	24
4.8.3	Data Flow Diagrams . . . . .	25
4.8.4	Decomposition . . . . .	26
<b>5</b>	<b>Implementation</b> . . . . .	<b>29</b>
5.1	Recommended Deliverable . . . . .	29
5.2	Coding Conventions . . . . .	29
5.2.1	Code Structure . . . . .	29
5.2.2	White Space . . . . .	29
5.2.3	Names and Variables . . . . .	31
5.2.4	Modules . . . . .	31
5.2.5	Portability . . . . .	34
5.3	In-Line Documentation . . . . .	35
5.3.1	Comments . . . . .	35
5.3.2	Telephone and Pencil Tests . . . . .	35
5.3.3	Module Header . . . . .	36
5.3.4	Module Separation . . . . .	36
5.3.5	Variable Descriptors . . . . .	37
5.4	Data Organization and Libraries . . . . .	37
5.4.1	Data Organization . . . . .	37
5.4.2	Libraries . . . . .	38
5.5	Summary . . . . .	38
<b>6</b>	<b>Test</b> . . . . .	<b>39</b>
6.1	Recommended Deliverables . . . . .	39
6.2	Why Test? . . . . .	39
6.2.1	Types of Testing . . . . .	40
6.2.2	Preparing for Test . . . . .	40
6.3	Top-down and Bottom-up Testing . . . . .	42
6.4	Software Testing Stages . . . . .	43
6.4.1	Unit or Module Testing . . . . .	43
6.4.2	Subsystem Testing . . . . .	43
6.4.3	System Testing . . . . .	43
6.4.4	Documentation Testing . . . . .	43
6.4.5	Acceptance Testing . . . . .	43
6.5	Testing Real-Time Systems . . . . .	44
6.6	Test Documentation . . . . .	46
6.7	Debugging . . . . .	46

<b>7</b>	<b>Operation and Maintenance</b>	<b>48</b>
7.1	Recommended Deliverables . . . . .	48
7.1.1	Who Needs Training? . . . . .	49
7.1.2	Training Approaches . . . . .	49
7.2	Manuals and Procedures . . . . .	51
7.3	Conversion . . . . .	53
7.3.1	Conversion Preparation . . . . .	53
7.3.2	Conversion Approaches . . . . .	53
7.4	Maintenance . . . . .	54
<b>8</b>	<b>Configuration Management</b>	<b>55</b>
8.1	Recommended Deliverables . . . . .	55
8.2	Why Configuration Management? . . . . .	55
8.3	What Is Configuration Management? . . . . .	56
8.4	Change Control . . . . .	58
8.5	Maintenance . . . . .	61
<b>9</b>	<b>Verification and Validation</b>	<b>62</b>
9.1	Recommended Deliverables . . . . .	62
9.2	Why Verification and Validation? . . . . .	62
9.3	Reviews and Audits . . . . .	63
9.3.1	Formal Reviews . . . . .	63
9.3.2	Informal Reviews . . . . .	64
9.3.3	Walkthroughs . . . . .	65
9.3.4	Inspections . . . . .	65
9.4	Validation Testing . . . . .	68
9.5	Debugging . . . . .	68
<b>10</b>	<b>Summary Example</b>	<b>69</b>
	<b>Appendix A: References</b>	<b>71</b>
	<b>Appendix B: Glossary and Acronyms</b>	<b>78</b>
	<b>Appendix C: Sample Sandia Module Header</b>	<b>83</b>
	<b>Appendix D: Control Structures</b>	<b>84</b>
	<b>Appendix E: Inspection Report Forms</b>	<b>85</b>

<b>Appendix F: Change Control Forms</b>	<b>91</b>
---	-----------

<b>Index</b>	<b>94</b>
--------------	-----------

### **List of Tables**

1	Example Project Management Aids . . . . .	8
2	Reference Requirements Methodologies . . . . .	14
3	Example Interactive Debuggers . . . . .	47
4	Change Table Sample . . . . .	58
5	Baseline Table Sample . . . . .	59
6	Change Control Software Tools . . . . .	60

### **List of Figures**

1	Typical Software Life Cycle . . . . .	4
2	Software Lifecycle Error Sources . . . . .	16
3	DFD 0 - Software Development Data Flow . . . . .	26
4	DFD 2 - Design . . . . .	27
5	Mini-Specification . . . . .	28

# Software Standards, Practices, and Conventions

## 1 Introduction

One of the many advantages hardware holds over software is the ease of statistical comparisons. It can easily be verified that from 1978 to 1985, computer capability at the weapons design laboratories increased twenty-fold, yet one can only guess at the subsequent increase in the amount of software developed and/or maintained. Nevertheless, the growing importance of software within the laboratories is now an accepted fact, as is, unfortunately, the number of problems associated with software development. Doing the software job right is increasingly important, and thus standards, practices, and conventions designed to ensure the quality of software need to be documented and provided to all individuals producing software.

### 1.1 Intent

In the spirit of Confucius [“The palest ink will outlive the strongest memory”], this manual defines and documents software standards, practices, and conventions, and thus provides an information baseline for the individual engaged in software development, maintenance, and management within Sandia National Laboratories. The document outlines the deliverables that many Sandians recommend for such activities. The document and the deliverables are not limited to “coding techniques;” rather, included are recommended software design, development, and management practices. A two-page outline for implementation of these practices on a one-person or small project is provided in section 10. Such projects will require some modification concerning the detail included in the deliverables (*e.g.*, a plan may be defined in a short memo), but the deliverables are still needed.

### 1.2 Environment

On a daily basis, personnel at Sandia National Laboratories create software with diverse characteristics and requirements. WR (War Reserve)

software required for embedded computer systems, accomplished almost entirely in assembly language, must have high reliability and perform within a real-time system. Large and complex computer codes for finite difference analysis are produced in FORTRAN 77, and maintained for long periods of time. Administrative software is primarily written in COBOL, following the Systems Development Methodology/Structured [SDM]. Reimbursable projects for the Department of Defense are written in C or Ada, and often have to meet software development standards required by outside agencies.

The Sandia personnel who develop or modify these diverse software projects, however, have several characteristics in common:

- They are few in number per project. About a dozen programmers/analysts is as large as any project usually extends, and one software professional per software project is not uncommon.
- They want to produce a quality software product.
- They are under schedule and resource constraints.

The last two characteristics appear to be at cross-purposes with each other. The schedule and resource constraints can pressure programmers to begin coding prior to signed-off system requirements and formally produced system designs. Similarly, these external constraints (or even programmer priorities) can cause an inadequate amount of time to be spent documenting a project. In the case of a one-person project, the professional may look upon himself as the primary user, and feel that little documentation is required for this restricted application. If a requirement for a change were to appear, he/she assumes he/she will be the one to accomplish it.

### **1.3 Applicability**

The manual is designed to be applied to any software developed or maintained either by Sandia personnel or by personnel under contract to the Laboratories. The guidelines are recommended practices and conventions to follow, whether the software involved is WR or non-WR, written in assembly language or higher level language, developed by one person or a reimbursable project software development team, and whether produced from new requirements or an existing code. For example, although a sample coding technique in the implementation section of the manual may be demonstrated using a particular programming language, the principle illustrated can be applied in any language.

The manual is a set of *guidelines*, not directives. Enforcement of an application of these guidelines must be established at the project level or individual level, with management support and promotion. Each individual project leader must make a conscious decision as to the degree these guidelines will be imposed on his/her own project, or the appropriate "tailoring" of these guidelines to the project and environment at hand. Additional references may be required to provide the technical details and design principles that create the successful design engineering environment. Once the standards for the project have been established, they must be publicized and rigidly enforced. This may be accomplished by referencing this document (or its applicable subsections or deliverables) in an Engineering Procedure, a Project Plan, or a Software Quality Assurance Plan.

#### **1.4 Organization**

Section 2 of the manual outlines characteristics of software project management and provides recommendations for project planning. The next five sections address five of the eight major phases of a typical software life cycle [Figure 1]: Requirements, Design, Implementation, Test, and Operation and Maintenance. Next are two sections which cover software quality assurance activities which occur throughout the life cycle: Configuration Management, and Verification and Validation. The last section provides a two-page summary example of how to implement these practices on a Sandia project.

References for additional information are listed by section in Appendix A. Appendix B provides a glossary of terms. For follow-on investigation, the book references are available through the Sandia library system, and DELTAK courses on structured analysis, design, programming, and techniques are available via the Sandia Computing Education Center.

#### **1.5 How to Use This Manual**

The manual was designed to aid in planning Sandia software tasks. The user can turn to any section and immediately apply the information provided. Each section begins with the Recommended Deliverables for that phase or that activity, each marked by a □ symbol and followed by information to define those deliverables and describe why they are recommended. The Summary Example (section 10) provides an outline for small Sandia software development projects. An Index at the end of the manual references both terms (*e.g.*, **portability**) and phrases (*e.g.*, **structured**

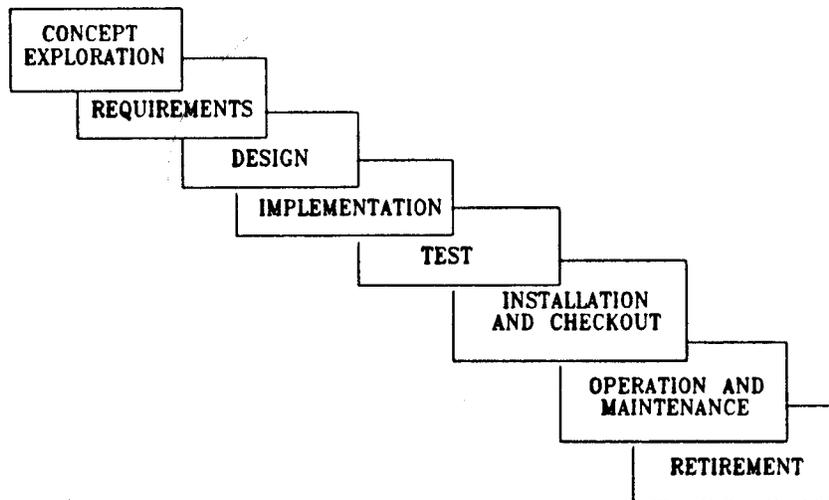


Figure 1: Typical Software Life Cycle

source code).

The manual is designed to be used in conjunction with references [SSGv1] through [SSGv5], which will provide details on software quality practices at Sandia National Laboratories, and reference [IEE84q], the IEEE standard for preparation and content of Software Quality Assurance Plans. The IEEE Guide [IEE85] gives some detail on these plans. The IEEE references are available through Sandia's Design Information Center, or by ordering from the IEEE Computer Society.

## 1.6 Summary

This manual documents software development practices that have been proven useful within the Laboratories and elsewhere, and presents a framework for creating a uniform product for follow-on software efforts. Tailor the application of these guidelines to a specific project based on its size, complexity, and criticality. Remember: on any software project undertaken, the software produced will most likely be modified by someone other than the original programmer; so it is important to create a quality product that any software professional would be glad to maintain. It has been said that "quality cannot be tested into software; it must be designed in." At Sandia, the goal is to engineer in software quality, review out software defects, and test out software errors. Following the guidelines as indicated will help achieve these objectives and produce quality software.

## 2 Project Planning and Management

### 2.1 Recommended Deliverable

Project Plan

- **Project Plan.** The project plan is a brief overview of the project. It defines the project, describes the organization, proposes schedules and milestones, and defines procedures to ensure the quality of the final product.

The consequences of neglecting the project plan include confusion over the deliverables, lack of understanding of the scope of the project, unclear requirements for resources and time, lack of unifying direction, and inability to determine project status.

### 2.2 Why Project Planning?

The goal of a software project manager is to produce a quality software product on time and within resource constraints. A quality product is one that meets the needs of the users and can be reliably used and easily maintained.

Project planning and management involve many activities. Initially the scope of the project must be determined; the deliverables must be specified; and a strategy for accomplishing the project must be outlined. Once this plan is approved by management and the user agrees that the list of deliverables is accurate, then work can begin. During the life of the project, the project manager is responsible for many tasks. These tasks include providing continuity between phases of a project and through personnel changes, foreseeing resources (tools, information, staff) that will be required and making them available, determining the status of the project, and revising schedule and budget estimates.

The project manager is responsible for assuring that the deliverables of the project are of high quality. This involves writing and administering a quality assurance plan. This plan describes the standards, practices, and conventions to be applied in the software project, configuration management practices, and validation and verification techniques. The outline for such a plan is provided in Volume 1 of these Guidelines.

The information in this overview of software project management can be found in many references. It is just a special case of technical project

management for which a huge body of literature exists. If there is one thing that Sandia excels at it is project management as applied to weapons (hardware) projects. Standards and practices are documented in the Engineering Procedures Manual. These same ideas can be applied to software projects as well.

Since many software projects involve very few people, some people may assume that the project management function can be ignored. This is wrong. The size and complexity of the project can determine the formality and depth of detail of the project plan and management practices. Project planning is important even on small, simple projects.

### **2.3 Project Plan**

The project plan is to be produced by the person with overall project responsibility. It is a changing document that is not complete until the project is over.

What follows is an outline for a project plan. A plan should consist of at least one sentence on each topic, even if it is "Does not apply because ...".

#### **Project Plan Outline**

- 1. Project Definition**

This is a 2-3 sentence statement of the project's overall goal.

- 2. Project Personnel**

Necessary functions and the people responsible for them are identified. If the project team is known, give names. If no staff has been assigned, describe the type of people required. If this is a one person project, list the functions that have to be performed with a notation of people to use as resources or reference.

- 3. Project Requirements**

This will become the Software Requirements Specification to be produced in the requirements phase. A rough draft of the specification is included here before the final version is approved.

- 4. Implementation Strategy**

This describes the plan-of-action used to accomplish the project.

- (a) Schedule/Milestones**

Major milestones and accomplishment dates are included here.

Dates are initially estimates that are revised and finally replaced by the actual date. A minimum set of milestones are:

- i. Start Date
- ii. Software Requirement Specification Approved
- iii. Software Design Description Approved
- iv. Software Test Plan Approved
- v. Implementation Completed
- vi. Testing Completed
- vii. Release

(b) Resource Plan

This describes the types of resources required by phase. Resources include staff time, hardware, and other items such as travel, software, and tools. Resources translate into budget.

(c) Deliverables

This will be a list of items to be produced by the project and the person primarily responsible for preparing them.

(d) Standards

Standards, practices, and conventions to be applied in the software project for requirements, design, coding, and testing.

(e) Dependencies

Other projects or conditions that this project requires for completion or success are listed. Alternatives are given.

5. Quality Assurance Plan

This section discusses how conformance to project requirements and standards will be evaluated.

6. Training

If project development personnel require special training (computer languages, software development techniques, etc.), the details of the training are listed: courses, schedules, trainees, and costs.

## 2.4 Project Management

Project management is the ongoing effort to guide, assist, and evaluate the project development task. It is a control mechanism. Information on status is achieved through frequent, measurable milestones (as often as every

two weeks). A milestone is an opportunity to assess progress and make changes if needed.

The project plan is constantly being revised to reflect the true state of the project. In addition to the project plan, a project file can be helpful. It is an organized collection of pertinent data related to the project. It is used for reference and communication.

During the course of the project development task, problems can occur. The impact of these problems needs to be assessed and resolved. It is a project management responsibility to bring important decisions to the appropriate level of management.

## 2.5 Tools, Techniques, and Methodologies

The tools mentioned below are not specific to software project management. They are general project management packages. Such software packages differ from each other in their capabilities, available output, and the complexity of the projects they can help manage. Regardless of the tool, users are still required to think. All of the packages listed below use one of two project management techniques or a combination of both. The techniques are PERT (and CPM) and Gantt charts.

PERT (Program Evaluation Review Technique) and CPM (Critical Path Method) are network planning concepts. Given information on activities, time constraints, material requirements, and critical resources, the user can obtain information on constraints, the critical path, and slack times. These techniques rely on network diagrams and automated tools to be manageable.

Gantt charts emphasize a visual representation of a project plan. These planning charts show start date and time to complete activities. Some of the information available from the PERT and CPM techniques are frequently integrated into the Gantt technique.

<i>Aid:</i>	<i>Technique:</i>	<i>Available on:</i>
TELLAPLAN	Gantt, PERT	VAX/VMS
Microsoft Project	Gantt	IBM PC/DOS
VisiSchedule	Gantt	IBM PC/DOS
Harvard Total Project Manager	Gantt, PERT	IBM PC/DOS
Diagram Master	Gantt	IBM PC/DOS

Table 1: Example Project Management Aids

## 3 Requirements

### 3.1 Recommended Deliverables

- Software Requirements Specification
- Software Requirements Review
- **Software Requirements Specification (SRS).** The SRS is a description of the external interfaces and essential requirements of the software in terms of functions, performance, constraints, and attributes. Requirements are objective and measurable. The SRS is concerned with what is required, not how to achieve it.
- **Software Requirements Review.** A review of the SRS document is performed by project members, users, and management. It verifies that the intent of the SRS is clear, the software proposed by the SRS is what is desired, and that the project can proceed to the next development phase.

Omitting the SRS may result in any of the following problems: building a system that is not what the users or sponsor wants; going over budget and behind schedule due to lack of a complete specification from which to estimate schedule and resources; developing software that does not integrate with existing software, hardware, or environment because design constraints were not understood; having design and testing problems since a definition of the software is lacking; and creating misunderstanding, confusion, and low morale among the project team members.

The consequences of skipping the Software Requirements Review are misunderstandings about what will be delivered because formal approval and signoff by developers, users, sponsors, and managers of the project is lacking.

### 3.2 Why Requirements?

The initial phase of a software development project is the requirements phase. The purpose of this phase is to document *why* this software project was initiated and *what* this software project is to accomplish. It consists of two major steps. At the end of each step a document is produced.

The first step in the requirements phase is the production of the software requirements specification (SRS). It is a description of the requirements that

the software must meet. To formulate the requirements, the environment in which the software will be used must be known. Information about the current operation, the users' needs and problems, constraints on the solution, and user expectations is collected. This information will be incorporated in the SRS in various ways.

Make every attempt to include a knowledgeable user on the team producing the SRS. A user can be a valuable source of information and can help to keep the SRS focused on user issues.

In the requirements phase, several alternatives are explored including off-the-shelf software. The requirements team is responsible for proposing the simplest, most cost-effective solution – not just a solution.

The SRS document includes a statement of purpose and scope for the software, background information that is required to understand the specification, the users' view of the functions that will be provided by the system, and the entire environment (hardware, software, interfaces, operation) in which the software system will exist. This document emphasizes measurable requirements – speed, quantity, and volume. The SRS provides a standard against which the software design and implementation can be measured.

The second step is to determine the adequacy of the SRS, via a Software Requirements Review. Participants in the review are the project team members responsible for design and test, users, sponsors, and management. Participants must reach a consensus and formally approve the SRS.

There is an overlap between the requirements phase and the design phase. This is beneficial as long as the two phases do not become concurrent. The overlap with the design phase can provide feedback on feasibility, resource requirements, and user interface issues.

### **3.3 Software Requirements Specification**

The SRS is an explanation of *what* the software does from the user's perspective, not *how* it will be done. This document provides the developers, users, sponsors, and management an opportunity to say exactly what this project is to accomplish. All of these people formally approve it. If there are disagreements, they should be resolved at this time. Changes at the requirements phase are very inexpensive and very easy compared to changes made during the test phase.

The approved SRS is a controlled project document. Changes to it are approved by another Software Requirements Review. All members of the project team have access to the current version of the SRS to prevent re-

design.

Even a tentative attempt at writing an SRS can yield benefits. A one sheet summary that is referenced and current is worthwhile. First and foremost, the SRS must be readable and understandable. Diagrams are used liberally.

The outline provided below can be modified to fit specific needs or circumstances. An SRS is applicable to every software project except those that will be thrown away within one month. These type of projects are very rare since people share programs, and program source is rarely thrown away.

A good guide to the SRS is the *IEEE Guide to Software Requirements Specification* [IEE84r]. It provides explanations and examples and is the source of the following:

### Software Requirements Specification Outline

#### 1. Introduction

(a) Purpose

The reason for writing the SRS is explained. The intended audience is defined.

(b) Scope

This section presents an overview of what is to be produced. The wording used is as specific as possible.

(c) Definitions, Acronyms, and Abbreviations

This section explains the terminology that is used in the SRS. A very wide audience of people has to read and understand it. Glossaries such as the *IEEE Standard Glossary of Software Engineering Terminology* [IEE83g] can define general terms.

(d) References

This is a bibliography of all documents mentioned in the SRS.

(e) Overview

This is a summary of what is contained in the SRS and its organization.

#### 2. General Description

This section is intended to provide a macro view of the project. It does not list specific requirements. Its purpose is to create a context in which the specific requirements can be understood.

(a) Project Perspective

This section describes how this project and its products fit with other projects and products. If it is independent, this is stated.

(b) Software Functions

The functions to be provided by the project software are summarized.

(c) User Characteristics

The end users of the system are identified. There may be several classes of users with different skills who require different types of information.

(d) General Constraints

This section provides information about conditions that will limit the software design.

(e) Assumptions and Dependencies

Possible changes to the constraints that would affect the requirements are listed.

3. Specific Requirements

This section contains all the information that the software designer requires to complete the software design. This section is the heart of the SRS.

(a) Functional Requirements

Basic actions that must occur in the software are given for each function.

(b) Performance Requirements

This section contains numerical requirements to be placed on the software such as number of simultaneous requests or transmission time given file size and system load.

(c) Design Constraints

Limitations and restrictions are given that result from standards and hardware.

(d) Attributes

Special circumstances such as the need to process classified data may add software requirements.

(e) External Interface Requirements

Requirements for interfaces with users, hardware, other software, environmental power, and communications are specified.

### 3.4 Software Requirements Review

The Software Requirements Review is a formal examination of the SRS document by project members, users, sponsors, and management. Its goal is to verify that the SRS document adequately and unambiguously describes the project requirements, and that those requirements are testable and easily traceable.

A set of criteria for evaluating the SRS is established. One possible set is: unambiguity, completeness, verifiability, consistency, modifiability, traceability, and usability during the operations and maintenance phase.

A Software Requirements Review Report is produced that indicates deficiencies in the SRS and written addenda or corrections that would resolve the deficiencies. All the participants must agree on the SRS. They must understand it, agree that it describes the software desired, and give authorization to continue development.

Guidance in conducting a formal review can be obtained in Section 9 of this document.

### 3.5 Prototypes

Prototypes have a place in the requirements phase of a software project. A prototype is a minimally functional system used to illustrate the software's user interface or to prove the feasibility of a concept. *Caution:* This is a non-standard definition of the word prototype. Engineers would typically think of a prototype as a first, fully functional system or a pattern. The term as used here is common in the software community, as in "rapid prototyping." Synonymous terms are façade, demonstration model, and cartoon. An analogy used to explain a software prototype is the Western town created for a movie, where buildings are just façades. The viewer sees only the street which appears realistic.

Prototypes are used to understand a complex user interface. The users and developers may not know exactly what the interface requirements should be. Prototypes allow both of them to get some experience with the user interface before the requirements are set. Users must be involved in a prototype if it is to be valuable. Human factors experts can be helpful in designing prototypes.

Another use for prototypes is to prove the feasibility of a concept. A simple model that requires minimal resources is created to learn about the system, try out new algorithms, verify capabilities, etc. It is this type of

prototype that can become *the* system. This pitfall must be avoided. It is necessary to clearly define the scope and limitations of the prototype. There must be a way to determine when the prototype is complete. Given the prototype experience, the SRS can be completed. There is no obligation to use any part of the prototype in the design or implementation.

### 3.6 Tools, Techniques, and Methodologies

The following methods are available to help define the user's requirements and produce the software requirements specification. These methodologies are not presented with any endorsement or suggestions. They are listed in Table 2 as references for further investigation.

<i>Acronym:</i>	<i>Methodology:</i>	<i>Reference:</i>
SSA	Structured Systems Analysis	[GAN79], [DEM78]
SADT	Structural Analysis and Design Technique	[ROS77], [ROS77b]
PSL/PSA	Problem Statement Language/ Problem Statement Analyzer	[TEI77]
SREM	Software Requirements Engineering Methodology	[ALF77]
USE	User Software Engineering Specification Method	[WAS79]
SDM/S	Systems Development Methodology/ Structured	[SDM], [SNL]

Table 2: Reference Requirements Methodologies

These approaches attempt to provide a structure for the specification and to illustrate the system development process. For a brief description of the first five techniques, see [WAS80]. Another source of information is Volume 5: Tools, Techniques, and Methodologies [SSGv5].

## 4 Design

### 4.1 Recommended Deliverables

- Design Description
- Design Review
- Design Review Results

- **Design Description.** A Design Description documents the design work accomplished during the design phase. Documenting the design prior to coding avoids (or reduces) any design misunderstandings and subsequent re-coding.
- **Design Review.** A Design Review is held to present and to discuss the design. The design is reviewed to discover any design inconsistencies or unmet requirements prior to the implementation phase.
- **Design Review Results.** The results of the review are documented in a report which identifies all deficiencies discovered during the review along with a plan and schedule for corrective actions. The updated design description document, when placed under configuration control, will establish the baseline for subsequent phases of the software life cycle.

People on projects who wish to avoid the design phase should be aware that most software life cycle errors occur during the requirements and design phases, as shown in Figure 2 below. If these errors are allowed to propagate through the implementation and test phases, they will be more costly to correct.

Tailor the design deliverables described in this section to the project at hand. For example, subsection 4.6 discusses two different design description documents which are produced at different points in the design process. A large project, involving many people, may require both documents. A smaller, or less complicated, project should need only one document. Similarly, a project with many interfaces between users or between different software may require the two reviews discussed in subsection 4.7. A project with few interfaces or easily controlled interfaces should require only one, possibly less formal, design review. Any structured walkthroughs may be delayed until the implementation phase when software coding is done.

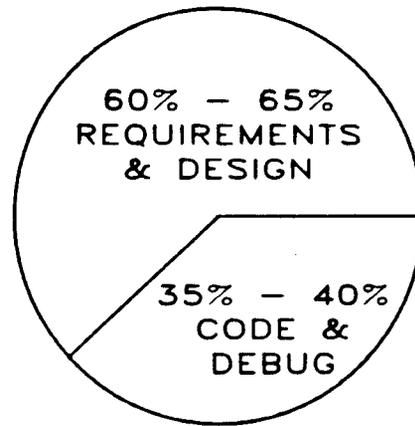


Figure 2: Software Lifecycle Error Sources<sup>1</sup>

## 4.2 Why Design?

The requirements and design are two of the most important and neglected phases in the development of software. If the requirements and design have been developed in an organized and systematic manner, then all that remains is for the program logic (described in pseudocode or structured English) to be translated into program code. Take the following to heart:

“Think first, code later.”<sup>2</sup>

Many coding problems can be avoided by following good design methods before doing any coding. Yourdon and Constantine [YOU79] emphasize that by “introducing a specific formal design activity to describe fully, and in advance, all the pieces of a system and their interrelationships, we have not created a new activity in the program development cycle. Structured design merely consolidates, formalizes, and makes visible design activities and decisions which happen inevitably – and invisibly – in the course of every systems development project. Instead of occurring by guesswork, luck, and

<sup>1</sup>Source: Seminar by George Tice, “Software Quality and Productivity Improvement,” SNLA, Feb 13, 1986.

<sup>2</sup>quoted by David King but originally found in L. B. Chumra and H. F. Ledgard’s *COBOL with Style: Programming Proverbs*, Rochelle Park, N.J. Hayden Book Co. 1976

default, these decisions can be approached deliberately as technical trade-offs."

Software that is to be of any long-term use to others, is designed to meet their needs. Users are included in the design of the product and, more importantly, in the definition of the requirements of the product. The time spent in design will be more than compensated for by software that is easier to maintain. There probably is more software being maintained or modified at Sandia than software being developed.

Another aspect of well thought out requirements and design is especially important within Sandia: Hardware is relatively inexpensive; person-time and person-costs are becoming the primary factors in projects, and thus methods that primarily involve people need to be streamlined. Contractors should have a set of guidelines to follow in writing codes for Sandia.

Other sources give reasons for design. Freeman [FRE80b] writes, "Perhaps the most important reason to design is that the creation of complex systems involves a very large amount of detail and complexity (*i.e.*, relationships of many sorts between many of the parts). If this complexity is not controlled, then the desired results will rarely be achieved. Design is the primary tool for controlling and dealing with this mass of detail and its attendant complexity. The regularity and structure of design methods and techniques serve to guide us through complex chains of reasoning where we might otherwise become lost." Two other reasons for design are to aid in the discovery of the underlying structure of the problem and to improve system quality. During design, there "is not yet a huge investment in code and detailed decisions which cannot be changed when an evaluation indicates that desired system properties are not being met. If reliability, useful user functions, modularity, and so on are not planned for before programming is begun, then generally they will be unobtainable." [FRE80b]

### 4.3 Software Design Criteria

Once the software requirements have been established, the design phase can proceed. The design phase consists of the preliminary design of the software system, through the detailed design, up to (but *not* including) the coding in the implementation phase. The preliminary design defines the major elements of the software, the interfaces between those elements, and the flow of information through the system. The detailed or critical design provides a blueprint for coding. It includes sufficient detail for someone other than the software designer to develop the resultant source code. Design is

an iterative process. Even after a design has been reviewed and approved (baselined), it still is subject to modification through the change control process.

Several criteria are followed for a good design. The following guidelines are given by Pressman [PRE82]:

- A design should partition the system into elements which perform specific functions. A design should be modular.
- A design should have a hierarchical organization to utilize the control among the software elements. A design should be refined in a top-down manner.
- A design should lead to modules or subroutines that perform independent functions.
- A design should be derived from information obtained during the software requirements process. The design process should be repeatable. Design needs to be more of a science and less of an art.

Real-time applications (that measure, analyze, and control real world events as they occur) should not be exempt from the design process. Response times and bounds on software execution speed will impose additional design constraints. If response times are not met in the testing phase, the code can be checked and time-critical sections can be recoded.

#### **4.4 General Design Standards and Guides**

This section lists guidance available in two IEEE documents, the IEEE Software Quality Assurance standard [IEE84q] and the recently approved IEEE Guide for Software Quality Assurance Planning [IEE85]:

- Prepare a Software Design Description (SDD) to describe the major components of the software design (to include data bases, diagnostics, and interfaces). The SDD should describe how the software will meet the requirements of the Software Requirements Specification. It also provides a decomposition of the system into its components.
- Seriously consider using graphical techniques, top-down design, and a program design language.

- State what standards, practices, and conventions will be followed in the detailed design of the program modules and their interfaces. Cover such areas as naming conventions and argument list standards.
- Hold a Preliminary Design Review (PDR) to evaluate the technical adequacy of the preliminary design of the software as outlined in a preliminary version of the Software Design Description.
- Hold a Critical Design Review (CDR) to determine how well the detailed software designs as described in the Software Design Description satisfy the requirements of the Software Requirements Specification.

The SDD mentioned in this section is split into the preliminary design document and the detailed design document (subsections 4.6.1 and 4.6.2).

The following subsections on design give additional detail and suggestions for implementing the IEEE guidelines.

#### 4.5 Detailed Design Procedure

The individual elements of the software system such as subroutines, functions, or procedures initially should be described with an English-language narrative explaining the processing function of the module. Then, a detailed design tool can be used to translate the language narrative into a structured description that gives all necessary procedural detail. The following three types of detailed design tools should aid the designer:

- graphical tools
- tabular tools
- language tools

Graphical design tools visually depict procedural detail. These tools include flowcharts<sup>3</sup> and box diagrams (also known as Nassi-Schneiderman charts or Chapin charts for their developers). Data flow diagrams can be

---

<sup>3</sup>An interesting opinion with regard to flowcharts: Brooks [3] tells us, "The flow chart is a most thoroughly oversold piece of program documentation...The detailed blow-by-blow flow chart is an obsolete nuisance, suitable only for initiating beginners into algorithmic thinking. When introduced by Goldstine and von Neumann, the little boxes and their contents served as a high-level language, grouping the inscrutable machine-language statements into clusters of significance."

used to depict the flow of data through the system. There also are tools for modeling data [MAR83]. Section 4.8 discusses design tools in more detail.

Tabular tools include decision tables and IBM IPO (input-processing-output) charts, sometimes referred to as HIPO charts. These are discussed in more detail in subsection 4.8.

Language tools include a program design language (PDL), or pseudocode. It is not necessary to use specific design languages. The designer may prefer a type of highly structured English using IF-THEN-ELSE constructs, REPEAT UNTIL, BEGIN and END, FOR and DO, VARIABLE, CASE, and DO WHILE statements. The detailed module description is another approach. Such a description includes all data input to and output from the module, a list of all modules which call this module (CALLED BY), a list of all modules called by this module (CALLS TO), and a description or purpose of the module. This module description can later become some of the module header, discussed in section 5.3.3. The CALLED BY and CALLS TO pieces of information are necessary in the design phase to explicitly describe all module interfaces. Since no code has been written at this point, there is no other source for detailed module interface information. If the software is re-designed at a later date, the design documentation is updated to reflect any changes which also will be reflected in the code. An automated design tool may help. At the very least, the design information can be entered on a computer and accessed via a powerful editor or word processor.

## **4.6 Design Description Documents**

### **4.6.1 Preliminary Design Document**

The **Preliminary Design Document** (PDD) gives an overall representation of the software system to be designed. It documents the software structure addressed during the preliminary design phase and is used as input to the preliminary design review.

Pressman [PRE82] suggests that the following topics be included in the document:

- System objective and software's role
- Interfaces among hardware, software, and humans
- Major software functions

- External files and databases
- Design constraints and limitations
- Reference documentation
- Design description. This includes descriptions of the data, the flow of information, and the interfaces within the software.

If vendor software will be used as part of the system, the vendor documentation should be referenced in the PDD.

Much of the information is derived from the Software Requirements Specification (SRS) developed during the requirements phase. The PDD takes the SRS as input and expands the information to a more detailed design description. In other words, the software system is taken from the point of *what* should be done, to the beginning of *how* it should be done.

#### 4.6.2 Detailed Design Document

As the preliminary design moves into the detailed or critical design, the detailed software system description is extended down to the module level. Pressman [PRE82] suggests that the preliminary design documentation be expanded to include the following topics in the detailed design document:

- Module descriptions. This includes a description of the process, an interface description listing all data input and output from a module (including argument list data, external I/O, and global variables), and interfaces with other modules (called by and calls to). The description should clearly describe the major tasks and processing that occurs within a module.
- File structure descriptions. This includes logical descriptions of the external files and the data records.
- Global data descriptions.
- A cross-reference between the requirements and the modules critical to implementation of the requirements.
- Test provisions and guidelines.

- Packaging and software transfer considerations. This may include high performance requirements or physical memory limitations which may cause modification in the design and a description of the operating system characteristics necessary to understand the design.

## **4.7 Design Reviews**

There are many different approaches to software design review. Generally, the preliminary design review and the critical design review are formal reviews. They require significant preparation and may involve a fairly large number of reviewers (maybe 8 to 12). The formal review can act as a scheduled milestone for large software development systems. A smaller software effort, involving 1 or 2 people, may have a less formal preliminary design review with fewer reviewers. The preliminary design review is important to projects of all sizes to raise design issues (but not to resolve them) early in the project's life cycle and to gain both management and technical visibility.

The informal review involves a smaller number of people (perhaps 2 or 3) and may run from impromptu get-togethers to the structured walkthrough or the inspection process, discussed in section 9.3.

### **4.7.1 Criteria for Design Reviews**

A software design review may include representatives of management, design, quality assurance, and the end-user community. A rule of thumb is to include the same number of reviewers as designers. The reviewers should take an adversary viewpoint but should remember that they are challenging the software system design and design approach and not the designers. A benefit of the design review process is the early discovery and correction of software defects and errors prior to code development. It is best to get the most experienced people available as reviewers to detect as many errors as possible, as early as possible in the software life cycle.

All groups concerned with the design should participate in the review. This could include representatives from testing, software quality assurance, software development, system design, and the user/requester community.

### **4.7.2 Preliminary Design Review**

The preliminary design review should emphasize traceability of the design to the software requirements, the practicality and maintainability of the design, and the adequate definition of the interface and data structure

descriptions. Other design approaches should have been considered and reasons given for their rejection. Alternatively, the selection criteria may be enumerated. If the chosen design approach fails, an alternative approach may be available. Software limitations should be realistic, be acceptable to the final users, and be consistent with the Software Requirements Specification from the requirements phase.

Document the results of the review in a report which identifies all deficiencies discovered during the review along with a plan and schedule for corrective actions. The updated design document, when placed under configuration control, establishes the baseline for the detailed software design.

#### **4.7.3 Critical Design Review**

The critical design review is a review of the detailed design of the software system prior to code development and implementation. It also is called the detailed design review. The detailed design is examined to assure that it will be easy to translate into computer code and that it satisfies the Software Requirements Specification. The module descriptions should not be ambiguous. The software design should be verifiable, consistent with other elements of the system, and well documented.

Document the results of the review in a report which identifies all deficiencies discovered along with a plan and schedule for corrective actions. The updated design document, when placed under configuration control, establishes the baseline for coding.

#### **4.8 Methodologies and Tools**

There are many tools available. An overwhelming compendium of 412 software life cycle tools is given in [DAC85]. Many tools may not meet the needs of the Sandia project. For software projects with multiple designers, users, or interfaces, the design tool should be automated. Design changes can then be incorporated quickly without the initial design becoming obsolete. In the absence of all other tools, a rigorous design methodology could be combined with a word processor to document the design. Other features of the methodologies and tools to consider include the following: ease of use, short learning curve, ready availability to designers, graphics, structure, lack of ambiguity, and straightforward translation from detailed design to code. The discussion that follows indicates a few of the methodologies and techniques. Names and references are sprinkled liberally throughout for the

interested reader.

#### 4.8.1 Structured Design Techniques

The proponents of structured design can be classified according to approach: functional decomposition or data structured design [KIN84]. The functional decomposition approach includes data flow approaches and hierarchical structure charts showing structural aspects of a *system*. The data structured design approach emphasizes the structure of the *data* being processed. The functional decomposition methods seem better suited to the overall system specification and design phase, while the data structured design approach is appropriate for the design of individual programs and subroutines. As time goes on, the two structured design approaches may integrate each other's philosophies into their own.

Page-Jones [PAG78] discusses and gives examples of structure charts, data flow diagrams, pseudocode, data dictionaries, and mini-specs. Structure charts depict the partitioning of a system into modules and show the hierarchy, organization, and communication interfaces between the modules. The data flow diagrams may be used as precursors to the hierarchical structure charts. The set of lowest-level data flow diagrams may be translated into structure charts in order to show the time sequence of the processes necessary for coding.

Yourdon and Constantine [YOU79] discuss structure charts and compare them to the more familiar flow charts. The flow chart shows a sequence of steps to be executed, or the flow of control. Structure charts, on the other hand, distinguish between control data and normal data in the system with different types of connecting arrows between the boxes. The structure chart shows hierarchy, or which functions are subfunctions of which other functions. The boxes in structure charts are a bounded group of program statements which can be referred to as a unit. The IBM HIPO (Hierarchical-Input-Process-Output) representation of inputs to outputs complements the structure charts. The HIPO chart for a module should have three columns – the INPUT parameters, the OUTPUT values, and the PROCESSES which give the relationship or transformation process between the INPUT and OUTPUT entries.

#### 4.8.2 Software Design Tools

Several functional decomposition and data flow tools are available. Your-

don has aggressively marketed the data flow diagram approach. Other data flow approaches are the Structured Analysis and Design Technique (SADT) developed by Ross of SofTech and the Improved System Technology (IST) product of McDonnell Douglas Automation developed by Gane and Sarson.

Some of the techniques have been automated and can be used on a computer. Tektronix has automated the data flow diagram approach using DeMarco's book as a guide. Teledyne Brown Engineering has developed the Technology for the Automated Generation of Systems (TAGS) which consists of the following four software packages: storage and retrieval, configuration management, diagnostic analyzer, and simulation compiler. TAGS is available for Apollo and VAX 11/700 computers; the Tektronix tool operates on a VAX. In many areas at Sandia, personal computers are prevalent. The following personal computer software packages can be used for software design:

- Excelerator
- AutoCAD
- Action Diagrammer

#### 4.8.3 Data Flow Diagrams

The following paragraphs give an example of Yourdon's data flow diagram and mini-specification approach as an example of a design and documentation technique. A discussion of the graphical technique is given by DeMarco [DEM78].

A data flow diagram (DFD) is a graphical technique for representing information flow. The data flow diagram is also known as a data flow graph or a bubble chart. Figure 3, labeled DFD 0, gives the top level of a data flow diagram. User input which is external to the system is depicted by a box. Processes are shown as circles, while the flow of data between the processes are indicated by the arrows. Figure 4 (labeled DFD 2) is a more detailed break-down of process 2 (Design) in DFD 0. The final figure (5) is called a mini-specification and gives additional detail on process 3 (detailed design) in DFD 2. The data flow numbering scheme gives the number 2.3 to the mini-spec, indicating that it is a further break-down of process 3 in the level 2 DFD.

The entire software system model can be depicted by a single bubble with arrows representing the input and output data. This top level diagram

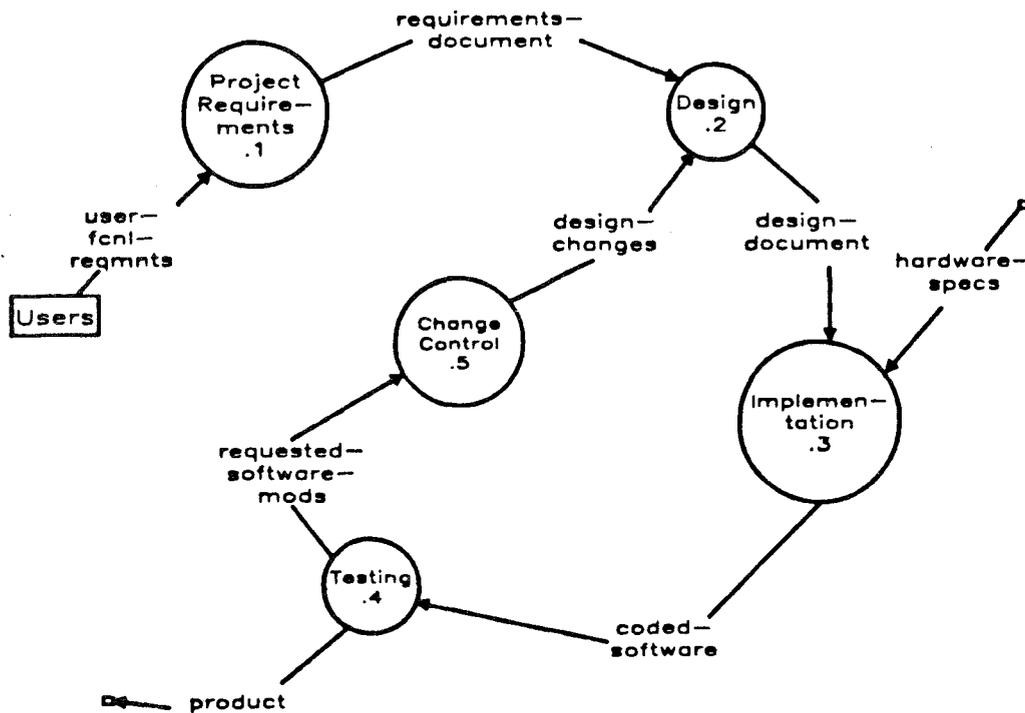


Figure 3: DFD 0 - Software Development Data Flow

can be refined into a series of bubbles to provide greater detail about the software system. The refinement can be continued to additional layers to show any desired level of detail.

#### 4.8.4 Decomposition

Myers [MYE78] discusses ways to decompose the design problem into modules. He defines a module as a group of executable program statements that are a closed subroutine and have the potential of being independently compiled and called from any other module in the program. The average module size should be about one page of executable statements. The module size may vary quite a bit, depending on the computer language used in the implementation phase and the internal complexity of the module. Each module should be highly cohesive (perform one single function) and

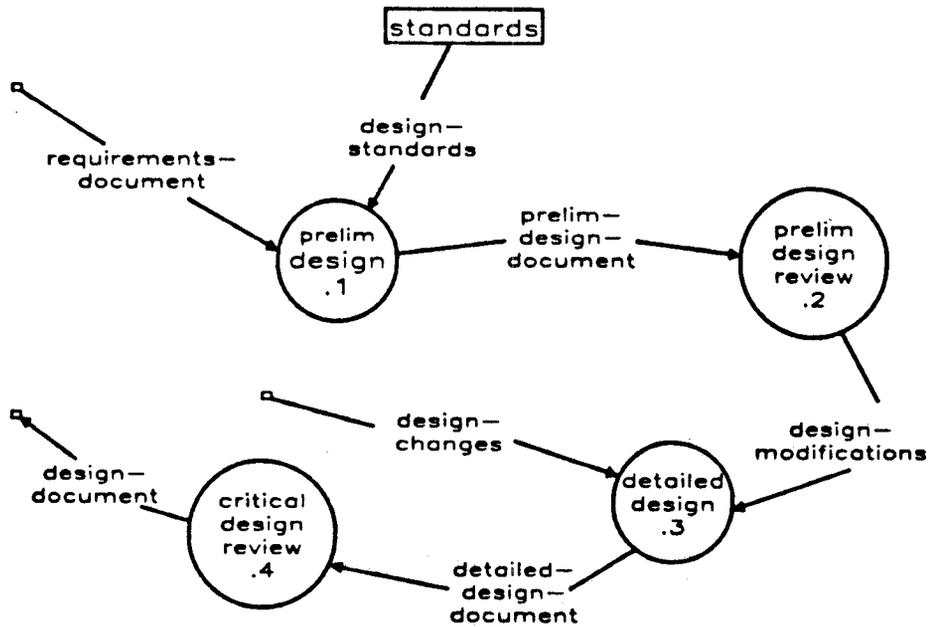


Figure 4: DFD 2 - Design

be loosely coupled to other modules (have few pieces of data passed between modules). As further reference, David King [KIN84] and Meilir Page-Jones [PAG78] discuss different types of cohesion and coupling in their books. The following rule of thumb also may prove useful in determining module size: if what a module does can be described by a simple sentence, then the level of decomposition is about right.

The methodologies and tools discussed here are merely examples of design tools available on both personal computers and mainframes. Volume 5 of these Guidelines will provide details on these and other software quality tools available to Sandia personnel.

```
detailed design
L.Marsupe
11/20/85
2 3

IF the design has gone through the change control process
    THEN incorporate the design-changes approved by the
         change control process
    ELSE incorporate the design-modifications into the
         prelim-design-document
END IF

DO (for all elements in the system structure)
    decompose the rudimentary system descriptions down
         to module descriptions
    write module descriptions listing inputs, outputs,
         and the purpose of the module
    create a detailed-design-document
END DO

END OF MODULE detailed design
[EOB]
```

Figure 5: Mini-Specification

## 5 Implementation

### 5.1 Recommended Deliverable

- Structured Source Code

Implementation is the translation of the detailed design into a computer language, a process commonly called *coding*. This section gives some suggestions on implementing good code. Well written source code is easier to read, test, debug, and modify. Many of the ideas in this section can be found in the book *The Elements of Programming Style* [KER74].

### 5.2 Coding Conventions

Once a software project detailed design has indicated a specific programming language, coding standards for that project must be established. The specifics of those standards are not as important as is their establishment and enforcement: *be consistent*. This section presents generic coding conventions, *i.e.*, concepts which apply to any language.

#### 5.2.1 Code Structure

Good code reads from top to bottom. Avoid “spaghetti code”, *i.e.*, any code in which the flow of control jumps around the source file so that program flow looks like a bowl of spaghetti. If the code is readable from top to bottom, less time will be spent interpreting program flow.

To aid in top to bottom flow use single entry, single exit control structures. Some examples are IF – THEN – ELSE, DO – WHILE, and REPEAT – UNTIL type structures. Appendix D presents graphical representations of these control structures. Some languages, such as assembler, do not provide these constructs or only provide a limited set. Missing high-level constructs can be implemented using more primitive constructs such as sequencing and conditional jumps. [BOH66]

#### 5.2.2 White Space

The use of white space (spaces, tabs, and carriage returns) can make code easier to read, and help show the logical structure of the code. Indent single entry, single exit control structures. Use blank lines when necessary to make the code less crowded. Consider the example on the next page.

```

/*****
function:      font

description:   Sets the fonts up.
inputs:       global variable with pointer to printer.
outputs:      sets primary or secondary font.
calls:        font_set. The actual work routine.
*/
font()
{
static char *m[]=          /* init. menu */
    {
    "Font Selection Menu",
    "Change Primary Font",
    "Change Secondary Font",
    "Return to Main Menu"
    };

int mr;
int ret=0;

do
    {
    do
        {
        mr = menu(m,3);      /* display the menu above.*/
        if( mr == -1 ) putchar(7); /* mr is selection number */
        }
    while( mr == -1 );

    switch(mr)
        {
        case 1: font_set("p"); break;
        case 2: font_set("s"); break;
        default: ret=1; break;
        }
    }
while( ret == 0 );
}
/*****/

```

### 5.2.3 Names and Variables

Use good mnemonics for names and labels. Some languages allow long names that can yield easy-to-read code. Use that valuable capability to its fullest extent. For example:

```
CYLINDER_VOLUME = CYLINDER_HEIGHT * PI * CYLINDER_RADIUS ** 2
```

Beware of languages that allow long names, but only use the first few characters for identification. The above example fails if the compiler uses the first eight characters to identify a variable name.

Some languages limit a name to a small number of characters (e.g. ANSI FORTRAN 77's limit is six). Good mnemonics can be generated by dropping letters from the end of a word:

```
CYLVOL = CYLHEI * PI * (CYLRAD ** 2)
```

Data must be mapped into the available data types in a given language. For example, temperature data might be mapped into real numbers. Some languages such as Ada provide a very rich and powerful environment for this task. Others such as LISP and FORTRAN provide a more primitive environment. The mapping should be done consistently throughout the program. Every variable may be explicitly typed or may be implicitly typed based on some special character in the variable name. Choose the mapping that yields the most readable and clear source code for the problem to be solved. For example, if a FORTRAN program requires only integers, then implicitly declaring all variables integers yields a wide-open environment for generating mnemonics.

### 5.2.4 Modules

Break programs into smaller, logically distinct pieces called modules. Modules are identified in the design phase of program development.

Keep modules small. Use module complexity as a guide to limit the size. Modules should have about one page of executable lines of code. Modules of one page (total) are convenient due to readability. Module size can vary depending on the language used.

Keep module interfaces simple. Pass the barest minimum of information to a module. *Keep local information inside modules.* When calling a routine,

the caller should pass only needed information, and the routine should return a well defined result. Pass information in a "need to know" manner.

Write modules with a single entrance and single exit unless violating this rule results in improved clarity or readability. Three commonly occurring situations where multiple entrances or exits are useful are multiple loop exits, error handling, and data encapsulation [FAI85]. Before using multiple entrances or exits, closely examine the algorithm and try to redesign it to eliminate the multiple entrances or exits.

The two following examples illustrate some of the code structures recommended in this section.

```

*****
* routine:    message
*****
*
* description: sends a message string to the console
* inputs:     message location offset on stack (word).
*             message of form (length,'message string').
* note:       length = message length + 1. (includes length byte)
*             max length of message 254.
*             This is a threaded routine, so no subroutine return.
* outputs:    none
* called routines: chout
*****
*
hmess    header    05,'mess',00,$0000
         move.w    (a5)+,a1        get offset from stack
         lea      (a4,a1),a0      compute mess address,put in a0
         clr.l    d6
         move.b   (a0)+,d6        mess length to d6
mtype    subq.b   #2,d6           fix count for dbcc
loop     move.b   (a0)+,d0
         jsr      $chout(a4)      chout is offset to output routine
         dbf      d6,loop         done outputting chars? no, loop
         next
         yes, end.
*
*****

```

```

C=====
C ROUTINE:      FGETC
C=====
C DESCRIPTION:  FGETC RETURNS THE NEXT CHARACTER IN A FILE.
C              EOF IS INDICATED BY CHAR(26): CONTROL-Z;
C              CR: CHAR(13) IS RETURNED FOR END OF LINE.
C
C * NOTE:      FGETC READS A LINE AT A TIME AND RETURNS NEXT CHARACTER.
C              DON'T USE IT FOR SINGLE CHARACTER TERMINAL INPUT.
C
C USAGE:       CHAR=FGETC(FILE_LOGICAL_UNIT)
C
C INPUTS:      LOGICAL UNIT OF FILE
C OUTPUTS:     NEXT CHARACTER
C
C VARIABLES:
C              CR      -   CARRIAGE RETURN CHARACTER
C              EOF     -   END OF FILE CHARACTER
C              LINE    -   CURRENT LINE OF TEXT
C              LP      -   POINTER INTO CURRENT LINE
C              LU      -   LOGICAL UNIT OF FILE TO READ.
C
C CALLS:       TRIM          GIVES LENGTH OF LINE WITHOUT
C                          TRAILING BLANKS
C ASSUMPTIONS: A SEQUENTIAL FILE HAS BEEN OPENED AS LU.
C=====
CHARACTER*1 FUNCTION FGETC(LU)
CHARACTER LINE*132
CHARACTER*1 EOF,CR
INTEGER LP
INTEGER TRIM
SAVE LINE,LP,I
DATA LP/O/
EOF=CHAR(26)
CR=CHAR(13)

C
C      WHENEVER THE LINE POINTER IS ZERO, READ A NEW LINE
C
C      IF(LP.EQ.O)THEN

```

```

        LP=1
        READ(LU,1,IOSTAT=I)LINE
        LENGTH=TRIM(LINE)
1       FORMAT(A)
        END IF
C
C * * Processor Dependent Code:  Interpreting non-zero IOSTAT:
C   I WILL BE -1 WHEN THE EOF IS REACHED, SO RETURN EOF CHARACTER
C
        IF(I.EQ.-1)THEN
            FGETC=EOF
            LP=0
C       WHEN LP PASSES THE END OF THE LINE, RETURN A CARRIAGE RETURN
C
        ELSE IF(LP.GT.LENGTH)THEN
            FGETC=CR
            LP=0
C
C       OTHERWISE RETURN THE CHARACTER AT LP
C
        ELSE
            FGETC=LINE(LP:LP)
            LP=LP+1
        END IF
        RETURN
        END
C=====

```

### 5.2.5 Portability

Whenever possible, work in a standard language (*e.g.* ANSI FORTRAN 77). Use of non-standard features which inhibit portability should be avoided. Portability (the ease of transferring software from one environment to another) makes code reusability possible. Isolate non-portable parts of the program in subroutines or functions and identify the non-portable code with comments. If a portable version of the code is available, then include it in the comments. Remember that any processor dependent code is non-portable.

Use symbolic names rather than explicit constants. For example:

```

C
C   EXAMPLE IN FORTRAN77 TO SHOW SYMBOLIC CONSTANTS
C
C   LUOUT AND LUIN ARE THE LOGICAL UNIT NUMBERS FOR THE CONSOLE
C   INTEGER LUOUT
C   INTEGER LUIN
C   PARAMETER (LUOUT=6,LUIN=5)
C*****

/* Whenever possible, parameterize the environment to enhance
portability of symbolic constants, as with the constant PI:
*/

    pi=4*ATAN(1)

/*****/

```

### 5.3 In-Line Documentation

#### 5.3.1 Comments

Many persons have stated "Good code is self-documenting". For most modern programming languages, however, this statement is false. Easy-to-read code can be written, but some in-line documentation is always necessary.

The number of comments to include in a program depends on the language used. Assembly language programs should have frequent comments, while high level languages such as Pascal may need comments only at module boundaries. When the code changes, update the comments so that they match the new code. Code maintenance is difficult enough without having the code and comments say two different things.

#### 5.3.2 Telephone and Pencil Tests

Two useful tests to check code readability and comments are the telephone and pencil tests. The "telephone test" checks for code clarity. If code can be read aloud (over the phone) and someone else can understand it, then it is clear. Otherwise rewrite the code. The "pencil test" checks for inline documentation. It consists of reading through the code, pencil in hand, and then including any notations made as comments in the next revision.

### 5.3.3 Module Header

Document modules with an identifying header. The following example gives a general idea of the format; a suggested format for Sandia is provided in Appendix C. Other information may be added if needed.

```
/*
*****

routine:      beep

description:  activates the terminal bell "rings" times
environment:  DEC VAX; VMS 4.1

usage:       beep(rings)

inputs:      rings  (Integer: # of times to activate the bell.)
outputs:     none
problems:    none known
assumptions: terminal has a bell
globals:    no global variables are changed
calls:      no user routines are called.
*/

beep(rings)
int rings;          /* number of times to ring bell */
{
    int i;
    for( i=1 ; i<=rings ; i++ )
        {
            printf("\007");
        }
}
/*
*****
*/
```

### 5.3.4 Module Separation

In the example above, the module starts and ends with a line of stars. This is one method of separating modules that appear in the same file. Use some consistent method to separate modules.

### 5.3.5 Variable Descriptors

Comment all variables individually. For example:

```
SUBROUTINE FPUTC(LU,C)

!!   THIS IS NOT FORTRAN-77

CHARACTER*1 C           !! PASSED CHARACTER TO SEND OUT
CHARACTER*132 LINE      !! LINE BUFFER FOR OUTPUT
CHARACTER*7 CCONT       !! CARRIAGE CONTROL RETURN VALUE
INTEGER LU              !! LOGICAL UNIT OF OUTPUT FILE
INTEGER LP              !! LINE POSITION POINTER
```

```
CYLINDER_VOLUME = CYLINDER_HEIGHT * PI * CYLINDER_RADIUS ** 2
```

If the language you use does not allow in-line comments, put the variable descriptors in separate comment lines as illustrated below:

```
C VARIABLES:
C           C --      PASSED CHARACTER TO SEND OUT
C           LINE --   LINE BUFFER FOR OUTPUT
```

## 5.4 Data Organization and Libraries

### 5.4.1 Data Organization

Many programs perform operations on enormous amounts of data. Some scientific programs are now performing calculations on hundreds of millions of words of data. Poorly structured programs will lump much of the data into large monolithic data structures. For example, FORTRAN programmers tend to lump data into a common block called *blank* common because of its special properties. Such a poorly designed "data organization by default" makes a program difficult to debug, modify, and maintain. In particular, it is impossible to communicate only the needed information to a subprogram (the need-to-know concept) if all the data is lumped together.

Apply the same modularity and design concepts used for developing subprograms to developing the data objects. Careful design of the modules in concert with careful study of the information flow will result in natural groupings of the data objects. Organize the data so these natural groupings can be accessed individually. This will result in modular data. With

modular data, a module can access only those portions of the data it needs (need-to-know). This makes checkout, debugging and modification much easier. If you wish to change a part of the data or some of the data is being destroyed, you can go right to the code that accesses the data.

#### 5.4.2 Libraries

Don't reinvent the wheel on every software project; take advantage of code reusability. Some languages provide features specially adapted to create reusable code, *e.g.*, Ada's packages and generics. A library of general purpose modules can save much time in coding an application, and reduce the testing time. On large projects libraries may be necessary. Even in small projects a library can be useful. Machine dependent routines are good candidates for libraries.

### 5.5 Summary

- Good code reads from top to bottom.
- Use single entry, single exit control structures.
- Use white space.
- Use meaningful names for labels and variables.
- Be consistent when mapping (data typing) variables.
- Use a consistent data mapping scheme that yields clear code.
- Break programs into page-long modules.
- Keep interfaces simple.
- Use single entries and exits for modules.
- Use multiple module entrances and exits  
only if clarity and readability are improved.
- Avoid language extensions.
- Update comments as code is changed.
- Use the telephone test for code clarity.
- Use the pencil test for inline documentation.
- Use consistent, descriptive module headers.
- Separate modules in a well defined manner.
- Describe all variables with comments.
- Organize all data.
- Use libraries for general purpose modules.

## 6 Test

### 6.1 Recommended Deliverables

- Test Set
- Test Set Documentation
- Test Results

- **Test Set.** The Test Set includes “rich” test data and relevant test procedures and tools to adequately test the application’s response to valid as well as invalid data.
- **Test Set Documentation.** The Test Set Documentation (or Software Test Plan) describes the test data, procedures, tools, and overall plan.
- **Test Results.** The results of the tests should be documented to identify all deficiencies discovered.

Without adequate documentation of the test set and results, the testing process may be disorganized, resulting in incomplete application testing and hampered application maintenance. Inadequate or nonexistent test data and associated documentation may result in inadequate exercising of changes made to the application. Without development and use of an adequate test set, the application may not meet the requirements specification.

### 6.2 Why Test?

Studies have shown that testing accounts for 30-50% of the total application development effort [SHO83], and currently is the primary method of determining whether a program or system does what it is supposed to do. While advances have been made in formal verification based on proof techniques, these techniques are tedious and are not easily applied without the help of automated tools such as verifiers. Additionally, a program’s “proof of correctness” is proving the provided input and output assertions, which may not be complete (see Gries, [GRI81] for these techniques). For individual programmers, testing continues to be the primary, if not the only, verification technique, and a vital, major part of the development effort.

Because of its importance and its potential impact on the development effort, testing requires careful planning and an understanding of what types of tests are applicable during the various stages of development. The following discussion expands on these topics.

### 6.2.1 Types of Testing

*There are two basic types of testing: dynamic and static.* [ICS82]

Static testing is evaluating a software program without executing it, whereas dynamic testing is based on execution of the program. Static testing spans each phase of the software life cycle. An example of static testing is the design review process. Static testing helps to ensure that the software meets project requirements and standards, and is of a sufficient level of quality to serve as a base for test planning. [EVA84] Examples of static testing are in the design and verification/validation sections of this document.

This section will concentrate on dynamic testing, which seeks to validate the program's function (black box testing) and to exercise its various paths and branches (also known as structure or white box testing). Volume tests and stress tests may also be useful [KIN84]. Volume tests involve providing more data than the system will normally handle; stress tests force the application to operate at its maximum rate. A great deal may be learned about an application by examining its response to these tests.

Volume and stress tests are applicable in both real-time and "normal" (delayable) applications. Some problems may not surface until a large amount of data is entered into the system. One example where volume testing was useful to one of the authors concerned finding a non-unique entry in a file index database. The probability of finding an error in the database schema was increased by introducing a large number of files.

### 6.2.2 Preparing for Test

*An important phase of testing is planning.*

Planning should include how to observe results, and compare the results with desired behavior. The comparison is not always a straightforward task. The system requirements specification (SRS) can help in this area. The SRS should provide a complete description of the application's behavior, the needed functions, and related constraints.

*Conduct module or unit testing.*

As pieces of the project are completed, these modules should be tested. (Please note that module and unit are used interchangeably in this section.) It is easier to locate errors when the number of possible interactions is limited to a small number of modules. Only after these modules have been debugged should they be incorporated into the program. This is the basic approach used in bottom-up testing discussed later in this section. If errors are found and modifications made, previous tests involving erroneous sections should be rerun. Testing is truly an iterative process. Therefore, ...

*Test the application early and often.*

Problems revealed early in the design process are orders of magnitude easier and cheaper to correct than problems discovered once the program is in production. Data structures and algorithms based upon design decisions are often subtly intertwined. Modifying the application's design to correct bugs late in the design process may involve a great deal of algorithm and data structure modification, which could introduce further bugs.

*A vital part of software testing concerns choosing data that will adequately test an application.*

The following guidelines discuss how to develop such a test set.

1. Exhaustive testing of all possible input values is not possible for many programs. Therefore, test data should be "rich": small enough to be manageable, yet comprehensive enough to cover the domain of input values. These "rich" test cases should contain data to exercise as many functions and traverse as many paths as possible. Minimally, each statement should be executed once during testing. Test data should also deal with boundary conditions.
2. The sooner the test data set is developed, the better. This can serve as feedback for other phases of the life cycle. For example, requirements for which it is difficult to formulate test data or determine the expected output are probably unclear and should be restated.
3. Test data should be documented as to what is being tested, and what the expected results should be. This set should be maintained for later use in checking out code modification.
4. Test data should include invalid as well as valid information.
5. Obtain some test data from the user community. Data that is typical of the application's ultimate environment should be included in the test set.

### 6.3 Top-down and Bottom-up Testing

There are two major testing approaches used in subsystem and module testing: top-down and bottom-up. These methods are normally used in conjunction with top-down and bottom-up software development, respectively.

Top-down testing starts at a high, subsystem level; subordinate routines are initially dummy routines. After subsystem testing, a module in the next level in the procedure tree is implemented and tested. This process continues until all modules in the subsystem have been implemented.

Bottom-up testing is the inverse of top-down; the functions in a module are tested. Modules are combined into a subset of the subsystem and tested. This cycle continues until the entire subsystem is built and tested. There are advantages and disadvantages with each approach. The advantages of top-down testing are:

1. Design errors may be identified more quickly, as these errors generally are inherent in the top portions of the system.
2. A demonstrable system is available early in the project, which may be good for morale as well as to prove/disprove feasibility of the system.
3. Major interface errors may be discovered earlier.

The disadvantages of top-down testing are:

1. It may be difficult to provide dummy routines that still adequately test the subsystem's functionality.
2. Test output may be hard to observe - many systems do not produce output in the higher levels of the subsystem.

The advantages and disadvantages of the bottom-up approach are the inverse of top-down testing. It is easier to observe a test's output, but no demonstrable working system is available until all the modules are tested and in place.

There is no *one* way to test an application. One approach may be superior in a particular application, but not in another. A combination of the two methods (sandwich) may be useful. This approach is predominately top-down, with bottom-up testing performed on some modules and subsystems. The development team can use both testing and integration techniques to their fullest advantage.

## **6.4 Software Testing Stages**

Software testing, like software development, should occur in stages. Each stage is a natural consequence of the previous stage. There are several distinct phases in the testing process.

### **6.4.1 Unit or Module Testing**

As a module is developed, its functions are tested to verify correct operation. Structure tests (traversing as many paths as possible) should also be performed. A good rule of thumb is to introduce modules incrementally.

### **6.4.2 Subsystem Testing**

Modules forming a subsystem are tested for correct cooperation and communication. At this point, an assumption is made about the correctness of the individual modules. Integration of units into the subsystem may be done using a bottom-up, top-down or "sandwich" (combination of top-down and bottom-up) approach. Careful planning is required to coordinate the development effort so units are available for integration when required.

### **6.4.3 System Testing**

This stage of testing is performed when all of the subsystems are integrated into the final system. Testing at this point focuses on locating design and coding errors undetected by prior design reviews and walkthroughs. Additionally, error recovery, throughput, capacity and timing considerations are examined. The system's operation is verified against the requirements specified in the system requirements specification.

### **6.4.4 Documentation Testing**

User guides should be tested for completeness and accuracy. The guides should be used during the system test phase before the application is turned over to users for acceptance testing [KIN84]. User guide examples of program operation or system function should be tested, and these test cases should be made part of the total test data set.

### **6.4.5 Acceptance Testing**

Acceptance testing involves testing the system with "real" data by the

organization who will be using the system. (Up to this point, the testing is usually performed by developers.) The application should be installed in the production environment for this stage of testing.

## 6.5 Testing Real-Time Systems

Programs in a real-time system present challenging testing problems that demand a higher testing standard. Real-time systems are those systems whose processes must respond to events under time constraints. If the system's response is not timely, information may be lost. Due to the types of applications for which real-time systems are developed, software errors can lead to disastrous consequences. Thus, the nature of real-time functions and their associated complex time-dependent interactions present additional testing problems involving more stringent timing and storage constraints. More intensive testing is needed to achieve a reliable operational status.

The attributes of real-time systems that complicate the testing effort can be summarized as follows.

1. **Magnitude of the programming effort** - Many real-time systems have a very large number of programs that have to be interconnected and tested.
2. **Repeatability** - Because of slight differences in timing, the same sequence of test case inputs, phased slightly differently each time, may result in different outputs.
3. **Equipment interaction** - Many real-time systems involve multiple processors that must exchange information. Development of the software for the differing processors typically is performed by machine oriented groups of personnel who work in semi-isolation to produce their machine peculiar subsystems. As two or more subsystems are tested and integrated, the lack of communication may be felt.
4. **Program interaction** - Several programs will typically share the computer at the same time. Without strict control of interfaces, significant errors can result because of unplanned or erroneous program interaction.
5. **Programming complexity** - Due to timing and execution constraints, real-time systems are frequently developed in assembly language. Software development is much more complex because of these constraints.

Extensive testing is required to eliminate programming errors. In addition, real-time systems will usually contain more decision points than batch oriented or scientific computation programs.

6. **Simulation** - Another consideration in the testing of real-time systems involves the credibility of testing in an environment other than the one in which the system will eventually operate. In many instances software development is done on "cross-compilers" or "cross-assemblers," where one processor produces executable code for a different processor. Module testing is often done on software which emulates the hardware environment in less than real-time, so timing problems are sometimes masked. Testing of real-time systems is normally first performed at a test facility using simulated input data before moving to the operational site. An extra effort is necessary to simulate with reasonable exactness the operating environment and "live" input.
7. **System operation** - A final attribute which complicates testing of real-time systems occurs after a system has been implemented. After a real-time system is operational, it becomes especially difficult to isolate and correct errors, thus the importance of adequate pre-installation testing is amplified.

The following phased approach could be used in conjunction with the testing methodology described elsewhere in this section to facilitate the testing of real-time systems. This approach is oriented around the validation of subsystems. Each subsystem consists of a set of programs that accomplish a single processing function. The testing begins with the exercising of individual subsystems one at a time and then progresses to testing multiple interacting subsystems.

- Phases I and II - Unit and subsystem testing (covered previously in this section). Because of the complexity level and asynchronous nature of real-time systems, fully testing new features and changes for undesirable side effects is very important.
- Phase III - Test the entire configuration (system testing). Simulated inputs are used to test more than a single subsystem at a time in order to test the subsystem interaction. A major objective of this phase is to stress the system and to determine its throughput capacity. This is accomplished by loading the system beyond its required capacity and observing whether the system takes the proper emergency measures.

Also, invalid data or messages are input to observe whether the system properly handles or rejects the invalid data or condition.

- Phase IV - The phase consists of a trial operational period at the operational site. This is required because it is very difficult (impossible) to create a simulated data environment that is identical to the operational environment and which can test every live data input condition. This testing phase is also an opportunity to conduct user training. This period concludes when the incidence of errors is reduced to an acceptable level and the system performs smoothly.

## 6.6 Test Documentation

The quality of test documentation can dramatically affect later stages of the software life cycle, particularly maintenance. Determining the amount of test documentation necessary for a particular project can be answered by the following questions: What would be of interest to me:

- *if I were managing a software project, and announcing its completion?*
- *if I were a new member of a "mature" software project?*

Unit testing normally is conducted by the program unit author. A test log during this phase of testing may be useful for later test phases.

System and acceptance testing requires documentation of the tests. Documentation may include a test plan, test design specifications, test case specifications, test log, test incident report or test summary report. The test case specification may include a test matrix listing application functions and paths and the data sets that test those features. General procedures and tools such as code profilers, test case generators, and driver programs useful in testing the application should be documented. The coding and documentation guidelines pertaining to structure, white space, names, and in-line documentation presented in subsections 5.2 and 5.3 should also be applied to test procedures and test data sets wherever possible to improve readability and maintainability. For additional information on test documentation refer to [SSGv2] and [IEE83t].

## 6.7 Debugging

Although debugging and testing are terms that are often used interchangeably, they are really distinct processes. Testing determines whether errors exist in an application; debugging diagnoses and fixes the errors.

The most primitive method of debugging is to print out the contents of memory at a particular time, *e.g.*, after a program abort or exit from a particular module. These are difficult to use because a variable's memory location must be identified, machine representations must be interpreted and the exact state of some machines (vector, multiprocessors) may be hard to define. If debugging must be done in a primitive environment, then you may want to design your modules with print statements of critical quantities. These print statements can be used or bypassed depending on an input parameter or enabled using compiler options.

There are several tools for static postmortem analysis. These tools print variable values after the program aborts. They are often symbolic in that they tell you the variable name and its value. DEBUG in the CRAY COS and Post Mortem Dump on CDC NOS operating systems print the variable values in the module that aborted. AUTOPSY on CTSS tells you where the program aborted and tries to tell you why. DDT on CTSS and DEBUG on VAX/VMS allow the user to look interactively at variables.

Interactive debuggers allow the user to trace execution, to display variable values, to stop the execution at breakpoints and generally to monitor and interrogate the state of a running program. A good symbolic interactive debugger can cut the debugging time by an order of magnitude. A symbolic interactive debugger allows you to refer to the variables by their names rather than their memory locations. The following table lists some interactive debuggers available on machines used at Sandia National Laboratories. Any serious software developer should become expert in the use of a symbolic interactive debugger.

<i>Debugger:</i>	<i>Available on:</i>
DDT	CTSS
DEBUG	ELXSI
DEBUG	VAX/VMS
DBX	UNIX 4.2
DEBUG	IBM PC (Professional FORTRAN)
CYBER Interactive Debug	CDC NOS

Table 3: Example Interactive Debuggers

## 7 Operation and Maintenance

### 7.1 Recommended Deliverables

- Maintenance Documentation
  - Training Plan
  - User's Manual/Operating Procedures
- **Maintenance Documentation.** Well documented code and the software design document provide the backbone of maintenance documentation and the starting point for determining training needs. Every software professional has been exposed to inadequately documented code, usually when trying to trace out the source of a problem.
  - **Training Plan.** The preparation of a well thought out training plan is an essential part of bringing a system into smooth operation. If the people, documents, and training techniques are not considered in the early planning for a new system, resources may not be available and training will be haphazard.
  - **User's Manual or Operating Procedures.** A user's manual is organized to contain practical information for the individuals required to put the software into action. Depending on the size and type of system, operating procedures may be required as a separate document to cover management of the logical and physical components. Without a properly prepared user's guide or operator instructions, either the time of the user will be wasted determining what to do, or the system will be inappropriately used, or both.

People are the key ingredient in any system. Providing adequate education initially, and on a continuing basis, is absolutely essential if a system is to achieve its objective. The alternatives for providing the training may be open to choice, the need for training is not. Depending on the project, a few pages of information, or a wide range of documents and approaches may be necessary. This chapter will present broad coverage of ideas for system operation and maintenance.

### **7.1.1 Who Needs Training?**

Three categories of people must receive some type of training in a new system.

The users are those who obtain a service from the software. These customers of the system may need an action, a computation, or a printed page. This is the group that should have been represented in the analysis activity of the requirements phase as needs were identified. These users now must be made aware of what the system requires, what it provides, and how it meets the identified needs. Acceptance testing is not complete until the customer is able to use the system to perform the design intent. Training can be a two-way street. The people using the software will provide some excellent feedback for system improvements.

The operating personnel are the second category. These are the people who will manage the system. They are involved with preparing input, processing data, and operating the logical and physical components.

A third group requiring training is the maintenance programmers. Even if the organizational philosophy is that the developers stay with the system, normal attrition will soon bring new people to the staff. Well documented code and the software design document provide the manuals for this training. A discussion of the need for in-line documentation appears in the chapter on implementation. Documentation tells how the design was translated to code and shows the expected result to be verified by testing. The close association of documentation with the source code has the advantage of making a complete story available in one place for maintenance personnel.

If the system is to receive the same level of support as the developer would give it, there needs to be a planned program for teaching subsequent maintenance programmers the basics of the business that the analyst learned when the project was initially studied. Too often it is assumed that the programmer understands the system simply because he knows how to make changes to the code. Maintenance programmers need information on the interrelationship of problems and needs that sparked the development project. They need to understand the technical, human, and organizational parts of the operational environment.

### **7.1.2 Training Approaches**

It is easy to calculate the costs of training. The time and materials used fit the budget mold. More difficult to calculate is the cost of insufficient

training. If a system is not utilized due to lack of understanding, the cost of the system is wasted. Probably the greater loss is the failure to make use of available information. Training is not something tacked on to the end of a project. The preparation of a well thought out educational plan is an essential part of the system implementation process.

To successfully provide people with the training required for the use and operation of a new system, it may be necessary to utilize several different approaches or combinations of them. These approaches may include:

- **Group Instruction** - This can be the best way to reach many people and provide them with an overview of the system.
- **On-Line Help** - The people who participate in group instruction may not use the system frequently. Through comprehensive on-line help this group can obtain satisfactory service. Another approach is the use of menus to move to the desired action. Menus are a burden to the frequent user, so there also must be a by-pass for the expert user.
- **Procedural Training** - In this method the individual is provided with written procedures describing the job tasks. This booklet may include a formal description of the system with detailed attention given to the outputs. To round out this training the individual would have an opportunity to ask questions of a trainer alone or in a group session.
- **Tutorial Training** - This technique provides personal training and may be necessary where many new ideas are introduced. It can provide one-on-one training for new user, operator, or program maintenance personnel. Although this can be fairly expensive, it assures the trainee will have a satisfactory understanding of the system.
- **Simulation** - A simulated work environment, using the data, procedures, and equipment involved, provides a medium for the individual to perform the proposed activities until a satisfactory level of competence is achieved. This technique would be used for operating personnel.
- **On the Job Training** - This is the usual method of training operating personnel. The individual is given simple tasks and specific procedures to start out. As these tasks are mastered, additional steps

are assigned. Although it has the appearance of providing immediate results or production, it can be a long and expensive approach.

- **Information Center** - The primary objective is to train existing personnel. However, training is rarely a one-time effort. Careful planning can result in a meaningful training mechanism which can be utilized by the organization on a continuing basis. This approach will justify the investment in more expensive aids and programs. The tools can be incorporated into an Information Center where consultants teach the users to help themselves. At such a center the user can get as little or as much training as is needed, and can get it immediately. The idea is as applicable to accounting and purchasing systems as it is to engineering information systems.
- **Train Trainers** - Another way to build for future training requirements is by training trainers. This will provide the select group of people who can deal with day-to-day problems. It creates more experts for the users or operating people to consult. It gives the system permanency by providing for employee turnover without making smooth performance dependent on the availability of the original analysts and programmers.

Regardless of the training approach selected, the effort should begin with a presentation of the overview. The overview can be the introduction of a document, the top screen of the menu set, or formal class presentation. Often training begins with a single task and then moves from task to task. The individual can better relate to the significance of each task and the process required if it is introduced by a system overview.

## 7.2 Manuals and Procedures

A user's manual is defined in NNWSI SOP-03-02 [SOP85] as "a manual which allows a peer to understand the results produced by the software, to run the software, and to install it on an appropriately equipped computer." This comprehensive definition accompanies a requirement for complete maintenance documentation which contains the theory of the original work, an in-depth explanation of the code, user/operator instructions, and a broad set of test problems. Certainly complexity, hazard, and liability are circumstances which would dictate the need for this type of coverage. However, the usual case is that the user's manuals are targeted to a particular level, group, and coverage.

User's Manuals and Operating Procedures are one way for management to exercise control over the activities of the organization. The purpose is to uniformly communicate what activities are to be performed, when, how, and by whom. The primary use of these documents is to assist in training. They also promote standardization and provide a guideline for system audit. The specific content of each procedure depends on the activity it describes. In general the following questions should be answered:

- What activity is being described?
- Why is the activity performed?
- Who must perform the activity?
- Where is the activity performed?
- When is the activity performed?
- How is the activity performed?

An excellent format for writing procedures is the Information Mapping Method (a registered trademark of Information Mapping, Inc.). Classes in the use of this method for report and procedure writing are regularly held at Sandia. This is the method now used in the preparation of the Sandia Laboratories Instructions (SLI's). Information Mapping uses the principles of how the human mind organizes information. The heart of the method is a component called the Information Block, which replaces the traditional paragraph. Each Block has a label that reflects the purpose and content of the Block. Standards for labeling, graphics, and formatting Blocks are an integral part of the method.

In compiling manuals, consideration should be given to providing a flexible format for content and update. If a user has no need for a complete set of system procedures, the manuals at a specific location should contain only the required procedures and an index to the complete set. A simple numbering system should be used to make it easy for users to update their holdings. Use of a loose leaf binder promotes both selectivity in content and ease in filing replacement pages. Weapon manuals are governed by detailed specifications for their construction, and can be a source of ideas for manual preparation.

### **7.3 Conversion**

Many software developments at Sandia have sought to automate manual methods or were motivated by advantages inherent in new technology. Conversion from an existing system to a newly developed system presents some interesting challenges. A primary concern in planning for training is the nature of the system being replaced. The success of a well designed and properly developed system may depend on how well the conversion is executed. When a new system produces inaccurate information, it can leave a mark that remains long after the problem is solved. The conversion must be planned carefully and woven into the training plan to avoid a credibility gap.

#### **7.3.1 Conversion Preparation**

Files are constructed during module and program testing to exercise the system without risking functional data. In the last stages of program testing a specific conversion plan for the existing data is prepared. It identifies any special start-up procedures, the schedule for file creation, acceptance criteria, and transfer of operating functions. It will be necessary to create files by collecting and organizing data in a specific format on a given storage medium. It will also be necessary to convert files by taking existing files and modifying them in format, content, and storage location. Elaborate control procedures may be required to ensure the integrity of the converted data. Assuring that all affected parties are aware of the procedure is a vital communications task.

A major system will probably involve three types of conversion: equipment, data processing method, and procedural. The change to improved equipment may not involve changing the logic of the application, but it will mean putting the logic in a coding structure which can be processed on the new computer. The data processing method can change from a manual or tape handling process to a terminal-controlled or computer-to-computer process. A procedural conversion can involve changing both the kinds of activities and the sequence in which the activities are performed.

#### **7.3.2 Conversion Approaches**

The three basic approaches for accomplishing the conversion to a new system are direct, parallel, and modular.

Direct conversion would be most applicable when the design of the new system is drastically different from the old system and comparisons between systems would be meaningless. This approach would also be used if the new system is small or simple.

Parallel conversion is the simultaneous operation of the old and new system. The outputs from each system are compared and reconciled. This approach provides a high degree of protection from a system failure. The obvious disadvantage is the cost of maintaining two systems. This means that the plan should call for periodic reviews with users and operating personnel so that reasonable criteria are set for stopping the dual systems, either to rework the new or to cut over to it.

Modular conversion refers to implementation on a piecemeal approach. This allows a partial commitment without affecting the entire operation. The piece could be the whole system in a particular locality or a part of the system installed across the board. This has the advantage of minimizing the risk of failure, but it is not always feasible because of the system, the organization, or the time it takes to complete the installation.

## 7.4 Maintenance

One of the realities at Sandia is that application programs, *e.g.*, scientific simulation codes, may evolve from other codes. Such programs can become unwieldy and difficult to maintain. Maintainers are adverse to removing unused code because they fear someday it may be required, or they are apprehensive of unseen coupling, or they are following the homespun wisdom, "If it ain't broke, don't fix it."

How to handle maintenance? Carefully – but apply these guidelines:

- Conduct an analysis phase (in lieu of the requirements phase) to:
  - determine what modules and documentation will be affected.
  - establish change control procedures.
  - establish standards, practices, and conventions.
- Where possible, re-write affected modules to bring them into compliance with the agreed upon standards, practices, and conventions.
- Where possible, follow the phases documented in these guidelines (*analysis*, design, implementation, test, operation and maintenance).

Section 8.5 provides a checklist for maintenance programmer responsibilities.

## 8 Configuration Management

### 8.1 Recommended Deliverables

- Configuration Management Plan
- Baseline Table
- Change Table

- **Configuration Management Plan.** The Configuration Management Plan lists all modules used by the project, module locations, personnel responsible for controlling changes and change procedures.
- **Baseline Table.** The Baseline Table lists modules and versions in the project's baselined system.
- **Change Table.** The Change Table lists all changes and enhancements made to the modules. Additional update supporting documents reflect changes and enhancements made to the system.

### 8.2 Why Configuration Management?

Changes must be managed right from the beginning of a project, whether developing a new system or modifying an existing one, working with a large team or alone. Any project evolves through additions to the system requirements specification document, the design reviews and changes in the structure of the design, coding development and testing, module testing, system testing and requests for changes and enhancements. Configuration management is the process of controlling these changes and enhancements.

The advantages of configuration management include

- protecting the interests of both the software developer and the end-users
- providing continuity to a project even as personnel change
- identifying a baseline which contains code and supporting documents at a specific point in time
- validating new baselines by independent audits.

A baselined system is the set of modules that has been system tested during development or is being used by the end-users. A configuration management plan describes the mechanism for establishing a new baseline and certifying the new baseline.

### 8.3 What Is Configuration Management?

Configuration management is the management of changes in the software (and associated documentation), and is applicable to the development activities as well as the maintenance activities. Managing changes to a software project may be easier for a single individual than for a team of people. However, being the sole developer and maintainer of the project has its pitfalls, the chief one being the tendency to ignore configuration management because all changes are funneled through that single individual. And, as the project expands or stabilizes, the individual must remember many details such as what changes or enhancements were made and why, which end-user has which version, which version of the modules is the latest, which versions were in the previous release of the system.

During the design phase, a project leader should implement a configuration management plan. This act forces discipline into the project by controlling changes. A configuration management plan may cover one or more phases of the software life cycle. Consequently, a project may have one or more plans or a plan with one section per phase. Before releasing the system to the end-user, the software development organization drafts the configuration management plan for maintenance, which is then approved by the end-user's organization, the QA organization for critical software, and other pertinent groups.

A configuration management plan should:

- identify the software files and the supporting documents;
- define the methodology for assigning and changing version numbers for both software and documents;
- define a procedure for each of the following activities:
  - a user requesting changes or enhancements
  - a programmer implementing changes to the software during the development or maintenance cycle
  - a librarian integrating software into the library

- illustrate the change or enhancement form and the response, disposition, or resolution report and explain the items listed on the forms;
- record release dates for new versions of software and documents and state the support termination date for older versions;
- describe audit controls for checking compliance with the plan;
- establish a procedure for archiving backups of master and versions, documents, and test cases.

The plan may need to address other items, dependent on the nature and scope of the software and the end-user's requirements. A more detailed, comprehensive guide can be found in the IEEE Standard for Software Configuration Management Plans [IEE83c].

Documenting changes made to modules and uniquely identifying each module is a must for everyone. Anyone can easily remember the minor changes made today, yesterday, or even last week, but six months later? The easiest place to document any change is in the source code itself. In addition, a change table is a useful tool for tracking unique modules by version numbers, dates, and the changes or enhancements made. The table is also called a project log, logbook, or a change document. Such a table should contain the module name, its version number, date of change, the responsible programmer, the locations of the change and a summary of the change. Table 4 provides sample entries from a change table.

A baseline table records each version of the system with all the modules that have been released or baselined. The table records the modules and their unique identifications, the date released, and the personnel who released the module. To identify each module in a released system requires accurate recordkeeping. The next generation of the system contains the latest changes or enhancements on some of the modules. After testing and verification of the test results, it will become a released or baselined system. Now, suppose some bug is found in the newly released system. It is desirable to find the version where the bug was introduced. Then, the bug can be corrected in the appropriate context or if possible, the system can be rebuilt temporarily with an older version of the module prior to the bug. An example of a simple baseline table showing five entries is in Table 5.

NCDRILL Log System				
Version	Date	Person	Program	Subroutine
1.002	09/10/85	K.T. Wilson	RETRNCINF	Put_land --Increase the format size from F10.4 to F12.4 for the x and y location on the error messages
1.101	10/25/85	K.T. Wilson	RETRNCINF	FIND_IDX, GET_IDX_TABLE_ENTRY --Increase maximum length of custodian field of the file index table from 2 characters to 3 characters
1.430	01/31/86	L. Jones	DRILLING	Get_title, Punch_driver, Append_comment, SNLA_write_holes, Tool_spindle --Lab requested a set of comment lines be added to the drilling holes file. The first line has the unique artwork number. Subsequent lines list each tool number and the drill size.

Table 4: Change Table Sample

Updating all the supporting documents is a very necessary part of any system. After every change or enhancement, the list of supporting documents should be reviewed to determine the impact of the change. A small change can easily require extensive editing and additional explanations in any or all supporting documents.

#### 8.4 Change Control

When a system is baselined, all the controls for changes must be rigorously enforced or else the system can easily degenerate into an unmaintainable, unreliable system. This does not mean that quick, emergency changes are not allowed. Rather, emergency fixes must be implemented as

CONFIGURATION ON THE CAD NETWORK			
Date	Person	Software	Version
11/04/85	K.T. Wilson	CHECKNC.EXE	1.102
01/20/86	K.T. Wilson	DRILLING.EXE	1.420
08/12/85	L. Jones	GERMASKS.EXE	1.010
02/04/86	K.T. Wilson	PANELPTS.EXE	1.000
02/06/86	K.T. Wilson	PANELPTS.EXE	1.100

Table 5: Baseline Table Sample

documented in the configuration management plan. Commenting or using distinctly formatted lines can easily identify the emergency fixes throughout the source code. Later, the problem is routed through the formal change procedure for proper analysis and appropriate action and a permanent change.

A precise method governing changes to the software should be thoroughly documented and distributed to all interested personnel and end-users. The project should assign one person to serve as the librarian. This is a necessary function and should be a part of any project. Smaller projects will not have a full time librarian, but the job must be done. The librarian controls all baselined versions of the system and adds the new modules or revised modules to the latest to-be-released system. Nobody else has the authority to add modules to a system. Ideally, the librarian is someone other than the code developers, a luxury that may not be available to small projects.

A project may have a change control board reviewing and approving the change or enhancement requests. The board may perform the independent audit for validating the modified software before that system is baselined. The board may have supervisors, quality assurance (QA) and other technical experts (including the project leader), and end-user representatives, depending on the nature of the project and the requirements or design specification documents. The configuration management plan defines the board's composition and functions. Fairley [FAI85] discusses the change control board and its functions in detail.

Change control can be done manually by adhering to strict record-keeping practices using the change table and the baseline table. Under

a manual system, the source code modules are still maintained in an electronic file. Start every such source code file with a header of introductory comments. This prologue is different from a module header as described in subsection 5.3. The source code prologue starts a file (which may include several modules). Include the following information in the header:

Name of Author:  
 Original Version Date:  
 Name of Revisor:  
 Revision History:  
 Name of Source Code File:  
 Names of any Include files:  
 Compiling Information:

Some of the above information may be inapplicable in some programs. In such cases use comments that are informative.

Several software tools are available for building libraries and tracing changes in the software and supporting documents. A brief survey of computer systems (Table 6) reveals some of the software packages available.

<i>Hardware</i>	<i>Software Tool</i>
CDC	UPDATE
CRAY	UPDATE
IBM PC/DOS	(none known)
IBM mainframe	LIBRARIAN, PANVALET
VAX/UNIX	SCCS, MAKE, RCS
VAX/VMS	DEC/CMS, DEC/MMS, EDCS, Softool
VAX/VMS and others	HISTORIAN

Table 6: Change Control Software Tools

Using prepared forms to request changes or enhancements to the software is the best method for controlling the changes to the software. At least the problem can be evaluated and the action and priority of the change documented. For forms to be effective a user should receive some response, either written or oral. Both the change or enhancement request and the evaluation and disposition report are under control of the responsible organization. If resources permit, problems or change requests may be reported

by using on-line forms. A database may be established that logs the problems and records the pending status and subsequent resolution. The first two entries in Appendix F illustrate change control forms used by Y-12 (Martin Marietta) [EDW85]. The third entry is an example of a form used at Sandia.

## 8.5 Maintenance

Most software engineering references agree that two-thirds of the lifecycle of software, in both time and resources, is spent on maintenance. Many Sandians can bear witness to the fact that for years they have worked only on maintaining software, never creating a software project from scratch.

Software maintenance is the process of working with operational software to correct errors and to provide enhancements. Software maintenance is often the most challenging job in the software life cycle when it should be the easiest. If guidelines such as those in this document have been not been consistently enforced during the development of the software, the maintainer's job is difficult.

Software changes or enhancements to a baselined module should be made only after careful analysis and approval by the project manager or the change control board. After the change is authorized, the following checklist can be used to assign maintenance personnel responsibilities:

- implement changes to the code
- update the internal documentation of the code
- validate the test sets in the baseline
- design new test cases as necessary, add the cases to the existing test set, and record the results on the new baseline
- revise the supporting documents (requirements, design specifications, test plans, user's manual, etc.)
- distribute the updated version of software and documents to the end-user sites and update the configuration control records on each site.

## 9 Verification and Validation

### 9.1 Recommended Deliverables

- Formal Reviews
- Informal Peer Reviews

- **Formal reviews.** Formal reviews should be held at the end of each phase in the software life cycle. The formal reviews should assess the compliance with previous life cycle phase requirements and products; satisfy the standards, practices and conventions of the phase; and establish the proper basis for initiating the next life cycle phase activities. [IEE85]
- **Informal Peer Reviews.** Informal peer reviews of the workproducts (*i.e.*, the documents or code) should be used during each phase of the software life cycle. The informal reviews should address the same topics as the formal reviews, but in much more detail and more frequently.

Recommended Deliverables for validation testing are listed in section 6.

### 9.2 Why Verification and Validation?

Verification and Validation activities are part of the quality assurance activities necessary to provide adequate confidence that the item or product conforms to established technical requirements [IEE83g]. Requirements are discussed in section 3.

Verification is the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase [IEE83g]. The phases in a typical software development cycle are shown in Figure 1.

Validation is the process of evaluating software at the end of the software development process to ensure compliance with software requirements [IEE83g]. Careful planning, design, and implementation substantially improve the probability of meeting the requirements. But, there is no guarantee the requirements will be satisfied. For example, you will not know if a real-time system will be fast enough without actually running it. Careful

testing, exercising and debugging of the actual software is required to validate the software. Refer to section 6 for more information on testing and debugging.

Boehm [FAI85] phrases the definitions as follows:

- Verification: “Are we building the product right?”
- Validation: “Are we building the right product?”

Note that verification activities can be applied to all phases of the software life cycle whereas validation activities are applied to the software produced by the implementation phase.

Verification and validation are used to determine workproduct conformance to specifications, detect defects as early as possible and improve the quality of the workproduct. Verification and validation activities use groups of people to analyze the workproduct and uncover defects. A software implementor can use systematic testing and debugging techniques to validate modules. Systems and subsystems can be validated by using formal validation and acceptance testing.

Verification and validation activities should be used as early as possible in a project. The earlier a defect is identified, the easier and cheaper it is to correct. An inconsistency between two models is much easier to correct in the design phase than in the testing phase. A defect in the coding is much easier to correct before copies of the software have been distributed.

### **9.3 Reviews and Audits**

Reviews and audits use knowledgeable groups to uncover defects in a workproduct. They are used throughout the software life cycle. The reviews may be formal or informal. Formal reviews or audits are important project milestones. They should be formalities that demonstrate that a major phase of the project has been completed. Informal reviews occur throughout the life cycle. They are the “inch pebbles” that build milestones.

#### **9.3.1 Formal Reviews**

Formal reviews or audits are formal presentations for the purpose of assessing consistency and completeness of the workproduct. The reviews should assess the compliance with life cycle phase requirements and products; satisfy the standards, practices and conventions of the phase; and

establish the proper basis for initiating the next life cycle phase activities. [IEE85]

Formal reviews involve authors of the workproduct and people who are not authors, such as users, management representatives, quality assurance representatives and consultants. A formal review should be held at the end of a phase in the software life cycle, *e.g.*, requirements specification (described in subsection 3.4), design (subsection 4.7), implementation, etc. Formal reviews can also be applied to critical parts of a project, *e.g.*, hardware-software interfaces. A formal audit or review should be a formality that demonstrates the informal audits and reviews have done their jobs, but at the same time provides management representatives a chance to review and provide comment.

### 9.3.2 Informal Reviews

Informal reviews and audits are used by the development team throughout the software life cycle. Members of the development team and others such as users, consultants and technical experts participate in the informal reviews. Management representatives do not participate in the informal reviews. The workproduct author uses a knowledgeable peer group to review his workproduct to assess compliance with items such as requirements, specifications, baseline standards, procedures, codes and contractual and licensing requirements. Informal audits and reviews are much more detailed than formal reviews. Informal reviews look at the workproduct line-by-line and try to identify individual defects. Walkthroughs and inspections are two procedures for informal reviews that are widely used in the software engineering community and will be described in the following subsections.

The informal reviews described below are consistent with the notion of egoless programming espoused by Jerry Weinberg [WEI71]. Egoless programming is the concept that workproducts are the responsibility of the entire group irrespective of who is the author or cognizant person for the workproduct. Informal reviews differ from the strictest interpretation of egoless programming because they may use people from outside the software development group.

The next two subsections briefly describe two types of informal reviews; walkthroughs and inspections. Walkthroughs have been used for many years and are well described in many software engineering books [SOM85]. Inspections are similar to walkthroughs but are more formal. They are newer and there is less information in the open literature [ACK84], [FAG76]. Because

of this, inspections will be covered in more detail than walkthroughs.

### **9.3.3 Walkthroughs**

Walkthroughs have been used for several years to systematically examine workproducts. The material being examined is presented by a reviewee and evaluated by a team of reviewers.

A walkthrough team usually consists of a reviewee and three to five reviewers. One of the reviewers may be designated moderator. The moderator's job is to keep the walkthrough focused on identifying defects. Team members may include the project leader, other members of the project team, a representative of the quality assurance group, a technical writer or other technical persons who have an interest in the project. Customers and users may be in the walkthroughs during the requirements and preliminary design phases, but are usually excluded from subsequent walkthroughs. Managers should not attend walkthroughs.

The workproduct should be distributed to the reviewers before the meeting so they will have time to become familiar with the workproduct before the meeting.

A walkthrough tries to discover defects by examining the product line-by-line. The reviewee is normally the author of the workproduct being reviewed. The reviewee reads and explains the workproduct and the reviewers try to identify defects or inconsistencies. Defects are not resolved during the session. It is the reviewee's responsibility to correct the defects. A follow-up meeting should be used to inform reviewers of the problem resolution.

Successful walkthroughs depend on establishing a positive, nonthreatening atmosphere for the session. The workproduct is being reviewed, not the reviewee. The moderator should keep the walkthrough focused on finding defects and not allow personality conflicts and minor problems to get out of hand. Set a time limit of less than two hours. Walkthroughs should never be used for employee evaluations.

More information on walkthroughs is available in [SOM85] or in the videotape [SDS].

### **9.3.4 Inspections**

An inspection is a peer review process for identifying defects in a workproduct. A peer group of three to six inspectors looks at a workproduct line-by-line, and identifies and categorizes defects. Inspections are similar

to walkthroughs, except inspections are more formal in structure and documentation. Inspections have formal entrance and exit criteria. Defects are classified according to type, class and severity. Documentation is generated for the individual inspection work, group work and follow up. The additional documentation allows trends in defects to be spotted. These trends may be related to the language used (*e.g.*, argument passing errors in FORTRAN) or programmer specific defects (*e.g.*, programmer B makes more typing errors than any other type).

### **Inspection Meeting Preparation**

An inspection must have well defined entrance and exit criteria. The entrance criteria for a software product may be a clean compilation and a successful pass through a standards and interface checker. The exit criterion normally is no defects.

Once an author believes his product satisfies the entrance criteria he (or someone in authority) selects an inspection moderator and completes the Inspection Profile form (Appendix E). The moderator's job is to check the workproduct for the entry criteria, decide whether or not to hold an overview, select the other inspectors, schedule the overview, schedule the inspection meeting, and be responsible for completing the Inspection Management Report and Inspection Summary Report (Appendix E).

An overview meeting may be required to provide the inspectors with needed background information. This presentation is normally given by the author.

The inspectors prepare for the inspection meeting by studying the workproduct and completing the Inspection Preparation Form. The author collects all material required for the inspection and distributes it to the inspectors. Each inspector should develop an understanding of the workproduct, note places where the understanding is incomplete and note places where the workproduct appears to have defects. Inspectors should defer detailed analysis and classification of defects until the meeting. The amount of time spent in preparing for the inspection should be approximately equal to the planned meeting duration. Each inspector should complete the Inspection Preparation Log (Appendix E).

### **Inspection Meeting Activities**

Defects are identified and classified in the meeting. The meeting should

have a well defined time limit that should not exceed two hours. The moderator reviews the inspectors' Preparation Logs and should hold the meeting only if the inspectors have adequately prepared for the meeting. The meeting should be postponed if the preparation time is insufficient.

Each participant in the inspection meeting has a well defined role. Everyone is an inspector. Some of the inspectors also have the duties of moderator, reader and recorder. The moderator and recorder may be the same inspector. The moderator must be able to guide the meeting, understand the goals of the meeting, and be an objective party. The reader is NOT the author. The reader paraphrases each "line" aloud for the group. The inspectors interrupt the reader with questions and concerns, and identify defects. The inspectors do not correct the defects, as this will be the author's responsibility. The recorder documents the defects in the Inspection Defect List. The author is also an inspector. He is an extremely important inspector because he knows the most about the workproduct history and structure. He should be responsive to questions and should not be defensive. Using a reader that is not the author is an important application of the egoless programming concept.

Defects are handled in a well defined manner. Defects should not be corrected during the inspection, although trivial defects such as typing errors may be resolved. Defects are categorized depending on the particular workproduct. In general, the type, class, and severity are defined. The defect type for a software product may be interface, data, logic, input/output. The defect class for a software product may be missing, wrong or extra. The severity may be major or minor. For a more complete discussion refer to [ACK84], [FAG76], or the videotape [FOW85].

### **Inspection Meeting Follow-up**

At the end of the meeting, the Inspection Defect List is reviewed and the workproduct disposition is determined. If the workproduct satisfies the exit criterion, then no rework or re-inspection is needed. If the exit criterion is not satisfied, then the group can decide if a rework or re-inspection is needed. If the defects are minor, the author may correct them and meet with the moderator. The moderator will check the rework and determine if the exit criterion is satisfied. If the defects are not minor, another inspection meeting may be required.

Besides completing the Inspection Summary, the moderator is responsible for the inspection meeting follow-up. He may have to verify corrections

for a rework.

The documentation is a very important part of the inspection process. By documenting and classifying defects, trends may be identified and statistics developed to guide future development efforts. Only the Inspection Summary is available to management. The other documents are used by the authors and their project leaders.

The Inspection Forms in Appendix E are for software inspections. Inspections can be applied to other types of workproducts such as program input decks or documents. The forms in Appendix E can easily be modified for different types, categories, and severities.

#### **9.4 Validation Testing**

Once written, a program may be validated by testing to ensure it satisfies its requirements, *e.g.*, correct results, required execution speed. A program unit should be tested by itself first (unit testing) and then integrated into the larger system and tested (integration and acceptance testing).

Tests cannot demonstrate the absence of errors, only detect their presence. In one sense, a successful test uncovers a defect. An unsuccessful test uncovers no defects. Testing cannot locate or correct the defect; debugging performs these functions. Section 6 provides details on testing.

#### **9.5 Debugging**

Debugging identifies and corrects source code defects. Defects or bugs will occur in even the best designed and implemented code, so they should be expected and planned for. For example, how many compilers do you know of that had no bugs when first released?

Debugging requires highly developed problem solving skills. The development environment strongly influences the amount of time required to identify a bug. Excellent interactive debuggers make identifying a bug much easier than inserting print statements. The software structure and design strongly influence the amount of software that must be modified to correct a bug. Good modular software using single entrance, single exit code structures and a modular data structure make modification much easier. For a more complete discussion of debugging, refer to subsection 6.7.

## 10 Summary Example

These guidelines provide a number of software standards, practices, and conventions that have proven useful in producing quality software. An example implementation of these practices is provided in the outline below. It has been used within Sandia for both in-house and contractor "small project" software development. Larger projects will require more formal procedures, *e.g.*, for verification.<sup>4</sup>

### Software Development Outline

1. **Requirements:** Determine the requirements of the new code or procedure to be developed.
  - (a) Write down the requirements.
  - (b) Discuss the requirements with the person responsible for the project (project leader) and potential users. Obtain formal approval to proceed.
2. **Design:** Construct a diagram to depict the flow of the data.
  - (a) Provide an annotated picture of the design.
  - (b) Discuss the design with the project leader.
  - (c) Ensure the design is consistent with the requirements.
  - (d) Generate a first cut at the module header documentation.
3. **Detailed Design:** Describe individual elements of the software system with a detailed English-language (pseudocode) narrative.
  - (a) Include enough details in the narrative so that coding would be easy to accomplish by someone other than the author.
  - (b) Ensure the narrative is easy to read. If the narrative becomes overly complicated, re-partition the design to make the overall flow easier to understand.

---

<sup>4</sup>For larger projects, the development process is broken into several tasks. Each task has a well defined workproduct. Each workproduct is inspected to ensure it satisfies the appropriate requirements. The discussions with the project leader in the outline may become formal audits that assess consistency and completeness of the workproduct.

- 
- (c) Discuss the narrative with the project leader. Ensure this level of design satisfies the requirements (step 1) and adheres to the data flow diagram (step 2). Project leader and programmer must both be convinced the algorithm described by the narrative will perform the desired functions.
4. **Test Preparation:** Construct a test set which will exercise the algorithm presented in the narrative.
    - (a) Develop a test set that is “rich”: small enough to be manageable, yet comprehensive enough to cover the domain of input values.
    - (b) Document the expected results of the tests.
    - (c) Note, discuss, and record any case that cannot be exercised due to difficulty in designing or implementing the test set, in case a problem arises later with that section of the code.
    - (d) Discuss the test set with the project leader.
  5. **Implementation:** Code the algorithm from the narrative.
    - (a) Include all appropriate documentation and commenting at this time.
    - (b) Walk through the code, checking that it performs the desired function.
    - (c) Guarantee that the code as implemented agrees with the algorithm specified in the narrative.
    - (d) Ensure the code is readable and maintainable by a programmer other than the author.
    - (e) Ensure the project leader is willing to maintain the code as written.
  6. **Validation:**
    - (a) Test the software on the test set.
    - (b) Investigate the possible implementation of any enhancements and future features. Document and re-test as required.
  7. **Operation and Maintenance:** Install the resulting production version of the code in the appropriate user area and place under configuration control. Arrange for user training.

## Appendix A

### References

#### 1. Introduction

*IEE84q* **The Institute of Electrical and Electronics Engineers, Inc.**  
*ANSI IEEE Standard for Software Quality Assurance Plans*, IEEE  
Std 730-1984, New York, 1984.

*IEE85* **The Institute of Electrical and Electronic Engineers, Inc.**  
*IEEE Guide for Software Quality Assurance Planning*, approved  
September 19, 1985.

***SDM* Systems Development Methodology**

For more information refer to SLI 1950 and Sandia National Laboratories' Computing Education Center Catalog.

***SSGv1* Sandia Software Guidelines**

Volume 1, *Software Quality*, SAND85-2344, Sandia National Laboratories, Albuquerque, NM, *expected printing* Jun 1987.

***SSGv2* Sandia Software Guidelines**

Volume 2, *Documentation*, SAND85-2345, Sandia National Laboratories, Albuquerque, NM, *expected printing* Jan 1988.

***SSGv4* Sandia Software Guidelines**

Volume 4; *Configuration Management*, SAND85-2347, Sandia National Laboratories, Albuquerque, NM, *expected printing* Jun 1988.

***SSGv5* Sandia Software Guidelines**

Volume 5, *Tools, Techniques, and Methodologies*, SAND85-2348, Sandia National Laboratories, Albuquerque, NM, *expected printing* Oct 1987.

#### 2. Project Planning and Management

***SSGv5* Sandia Software Guidelines**

Volume 5, *Tools, Techniques, and Methodologies*, SAND85-2348, Sandia National Laboratories, Albuquerque, NM, *expected printing* Oct 1987.

### 3. Requirements

**ALF77 Alford, M.W.**

"A Requirements Engineering Methodology for Real-Time Processing Requirements." *IEEE Transactions on Software Engineering*. SE-3(1):60-69;1977.

**DEM78 DeMarco, T.**

*Structured Analysis and System Specification*. New York: Yourdon;1978.

**GAN79 Gane, C.; Sarson, T.**

*Structured Systems Analysis*. Englewood Cliffs, NJ: Prentice-Hall;1979.

**IEE83g The Institute of Electrical and Electronic Engineers, Inc.**

*IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 729-1983, February, 1983.

**IEE84r The Institute of Electrical and Electronics Engineers, Inc.**

*ANSI IEEE Guide for Software Requirements Specifications*, ANSI IEEE Std 830-1984, New York, 1984.

**ROS77 Ross, D.T.**

"Structured Analysis (SA): A Language for Communicating Ideas." *IEEE Transactions on Software Engineering*. SE-3(1):16-33;1977.

**ROS77b Ross, D. T.; Schoman, Jr., K.E.**

"Structured Analysis for Requirements Definition." *IEEE Transactions on Software Engineering*. SE-3(1):6-15;1977.

**SDM Systems Development Methodology**

For more information refer to reference [SNL] and Sandia National Laboratories' Computing Education Center Catalog.

**SNL Sandia National Laboratories**

Information Systems Standards, *Sandia Laboratories Instruction*, SLI 1950, June 1979.

**TEI77 Teichroew, D.; Hershey III, E.A.**

"PSL/PSA: a Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." *IEEE Transactions on Software Engineering*. SE-3(1):41-48;1977.

**WAS80 Wasserman, A.I.**

"Information System Design Methodology" *Journal of the American Society for Information Science*, Vol. 31, No. 1, January 1980.

**WAS79 Wasserman, A.I.; Stinson, S.K.**

"A Specification Method for Interactive Information Systems." *Proceedings: Specifications of Reliable Software*. IEEE Computer Society; 1979: 68-79.

#### 4. Design

**DAC85 Data and Analysis Center for Software**

*Software Life Cycle Tools Directory*, Rome Air Development Center, Griffiss AFB, New York, Mar 1985.

**DEM78 Demarco, Tom**

*Structured Analysis and System Specification*, New York, Jun 1978.

**FRE80 Freeman, Peter**

"The Context of Design," *Tutorial on Software Design Techniques*, 3rd edition, New York: IEEE Computer Society; 1980.

**IEE84q The Institute of Electrical and Electronics Engineers, Inc.**

*ANSI IEEE Standard for Software Quality Assurance Plans*, IEEE Std 730-1984, New York, 1984.

**IEE85 The Institute of Electrical and Electronic Engineers, Inc.**

*IEEE Guide for Software Quality Assurance Planning*, approved September 19, 1985.

**KIN84 King, D.**

*Current Practices in Software Development - A Guide to Successful Systems*, Yourdon Press, 1984

**MAR83 Martin, James**

*Managing the Database Environment*, Englewood Cliffs, NJ, Prentice-Hall, 1983.

**MYE78 Myers, Glenford J.**

*Composite/Structured Design*, Van Nostrand Reinhold Company, Dallas, 1978.

**PAG78 Page-Jones, Meilir.**

*The Practical Guide to Structured Systems Design*, Yourdon Press,  
New York, 1978

**PRE82 Pressman, Roger S.**

*Software Engineering: A Practitioner's Approach*, McGraw-Hill, 1982.

**YOU79 Yourdon, Edward and Larry L. Constantine**

*Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, 1979.

## 5. Implementation

**BOH66 Bohm, C. and G. Jacopini**

"Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules," *Communications of ACM*, vol 9, no. 5, May 1966.

**FAI85 Fairley, Richard E.**

*Software Engineering Concepts*, McGraw-Hill Book Company, New York, NY, 1985.

**KER74 Kernighan, B.W., and Plauger, P.J.**

*The Elements of Programming Style*, McGraw-Hill, New York, 1974.

## 6. Test

**EVA84 Evans, M.**

*Productive Software Test Management*, John Wiley and Sons, 1984.

**GRI81 Gries, D.**

*The Science of Programming*, Springer-Verlag, 1981.

**ICS82 Integrated Computer Systems**

*Structured Design and Programming*, course notes, 1982.

**IEE83t The Institute of Electrical and Electronics Engineers, Inc.**

*IEEE Standard for Software Test Documentation*, IEEE Std 829-1983, New York, 1983.

**KIN84 King, D.**

*Current Practices in Software Development - A Guide to Successful Systems*, Yourdon Press, New York, 1984

**SHO89 Shooman, M.**

*Software Engineering: Design, Reliability and Management*, McGraw-Hill, 1983.

**SSGv2 Sandia Software Guidelines**

Volume 2, *Documentation*, SAND85-2345, Sandia National Laboratories, Albuquerque, NM, *expected printing* Jan 1988.

## **7. Operation and Maintenance**

**NUR83 U.S. Nuclear Regulatory Commission**

NUREG-0856, *Final Technical Position on Documentation of Computer Codes for High-Level Waste Management*, Division of Waste Management, Office of Nuclear Material Safety and Safeguards, Washington, D.C., Jun 1983.

**SOP85 U.S. Department of Energy**

SOP-03-02, *Software Quality Assurance*, NNWSI, Nevada Operations Office, Las Vegas, NV, *draft* Nov 85.

## **8. Configuration Management**

**DUN82 Dunn, Robert and Ullman, Richard**

*Quality Assurance for Computer Software*, New York, NY, 1982.

**EDW85 Edwards, J.E., Hebert, J.J., and Herr, C.P.**

Martin Marietta, *Software Development Standards for Martin Marietta Energy Systems Computer Applications Engineering*, Jan 1985, K/D 5391 R2.

**FAI85 Fairley, Richard E.**

*Software Engineering Concepts*, McGraw-Hill Book Company, New York, NY, 1985.

**IEE83c The Institute of Electrical and Electronic Engineers, Inc.**

*IEEE Standard for Software Configuration Management Plans*, IEEE Std 828-1983, New York, 1983.

## **9. Verification and Validation**

**ACK84 Ackerman, A. F., and Fowler, P. J.**

"Software Inspections and the Industrial Production of Software," in *Software Validation*, H. L. Hausen (editor), Elsevier Science Publishers B. V. (North-Holland), 1984.

**FAG76 Fagan, M. E.**

*Design and Code Inspections to Reduce Errors in Program Development*, IBM Systems Journal, Number Three, 1976.

**FAI85 Fairley, Richard E.**

*Software Engineering Concepts*, McGraw-Hill Book Company, New York, NY, 1985.

**FOW85 Fowler, P.J. and Ackerman, A.F.**

*Videotape of a Sandia Labs Presentation of Inspection Techniques by P. J. Fowler and A. F. Ackerman*. Refer to Nov 1985 issue of Sandia Computing Newsletter.

**IEE83g The Institute of Electrical and Electronic Engineers, Inc.**

*IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 729-1983, February, 1983.

**IEE85 The Institute of Electrical and Electronics Engineers, Inc.**

*Draft Standard for Software Verification and Validation Plans*, IEEE Computer Society, Dec 2, 1985

**SOM85 Sommerville, I.**

*Software Engineering*, 2nd edition, International Computer Science Series, Addison Wesley Publishing Company, Workingham, England, 1985

**SDS Structured Design Series**

DELTAK course; for more information, refer to Sandia National Laboratories' Computing Education Center Catalog.

**WEI71 Weinberg, G.M.**

*Psychology of Computer Programming*, Van Nostrand Reinhold, New York, N.Y., 1971.

## Additional References

1. **Boehm, B.W.**

"Software and Its Impact: A Quantitative Assessment" *Datamation*, Vol. 19, May 1973.

2. **Branstad, M., Cherniavsky, J., and Adrion, W.,**  
 "Validation, Verification, and Testing for the Individual Programmer",  
*Computer*, December 1980.
3. **Brooks, Frederick P., Jr.**  
 "The Mythical Man-Month," *Essays on Software Engineering*, Addison-Wesley Publishing Co., Menlo Park, CA, 1975.
4. **Connell, John and Brice, Linda**  
 "Practical Quality Assurance", *Datamation*, March 1, 1985.
5. **Conway, R., Gries, D., and Zimmerman, E.**  
*A Primer on Pascal*, Winthrop Publishers, 1976.
6. **EDP Analyzer**  
*Speeding Up Application Development*, Vol.23, No.4, April 1985.
7. **Freeman, Peter and Wasserman, Antony I.**  
*Tutorial on Software Design Techniques*, 3rd edition, New York: IEEE Computer Society; 1980.
  - (a) Boehm, B.W.: "Software and Its Impact: A Quantitative Assessment"
  - (b) Freeman, P.: "A Perspective on Requirements Analysis and Specification"
  - (c) Lundeberg, M.: "An Approach for Involving the Users in the Specification of Information Systems"
  - (d) Wasserman, A.I., Stinson, S.K.; "A Specification Method for Interactive Information Systems"
  - (e) Wasserman, A.I.: "Information System Design Methodology"
8. **Jensen, R.W. and Tonies, C.C.**  
*Software Engineering*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1979.
9. **Myers, Glenford J.**  
*Reliable Software Through Composite Design*, Petrocelli, 1975.
10. **Squires, R.**  
 Presentation notes, "Prototyping Beyond the Concept," *Cullinet User Week '85*, PRES-U008-UW85.

## Appendix B

### Glossary and Acronyms

Where possible, definitions in this glossary are taken from the *IEEE Standard Glossary of Software Engineering Terminology*, [IEE83g]. They are included here to provide a single-source document for the reader.

- **acceptance testing:** Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system. [see **system testing**]
- **algorithm:** A set of well-defined rules that gives a sequence of operations for performing a specific task.
- **ANSI:** American National Standards Institute
- **baseline:** A product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.
- **bottom-up integration:** A system of integrating modules in a program that starts with the bottom level modules and successively combines them to form larger systems. [See **top-down** and **sandwich integration**.]
- **boundary condition:** Extreme values (legal minimum/maximum for application), values falling on/near stated limits, special values (dependent on the application - *e.g.*, blank, negative, zero)
- **cohesion:** The degree to which the tasks performed by a single program module are functionally related. [contrast with **coupling**]
- **coupling:** A measure of the interdependence among modules in a computer program. [contrast with **cohesion**]
- **configuration control:** The process of evaluating, approving, or disapproving, and coordinating changes to configuration items after formal establishment of their baseline.
- **cpu:** central processing unit

- **debugging:** Debugging and testing are distinct processes. Testing identifies faults; debugging locates, diagnoses, and corrects the fault. Debugging's input is testing activity's output. [contrast with **testing**]
- **design review:** A formal meeting at which the preliminary or detailed design of a system is presented to the user, customer, or other interested parties for comment and approval.

**preliminary design review (PDR) :** The preliminary design review should emphasize traceability of the design to the software requirements, the practicality and maintainability of the design, and the adequate definition of the interface and data structure descriptions.

**critical design review (CDR) :** The critical design review is a review of the detailed design of the software system prior to code development and implementation. It also is called the detailed design review.

- **detailed design:** The process of refining and expanding the preliminary design to contain more detailed descriptions of the processing logic, data structures, and data definitions, to the extent that the design is sufficiently complete to be implemented.
- **field or operational testing:** Testing performed by the end user on software in its normal operating environment.
- **IEEE:** The Institute of Electrical and Electronics Engineers, Inc.
- **input and output assertions:** Statements, usually stated formally in terms of first order predicate logic, that describe what is true before and after execution of some piece of code.
- **inspection:** A workproduct review process where a reader reads through the workproduct and a group of inspectors try to identify defects. Similar to walkthroughs, inspections are more formal in structure and documentation. [see also **walkthrough**]
- **installation testing:** The formal process of confirming that a system or computer program is capable of satisfying its specified requirements in an operational environment. [see **field testing**]
- **integration:** The process of combining software elements, hardware elements, or both into a system.

- **integration testing:** An orderly progression of testing in which software elements, hardware elements, or both, are combined and tested until the entire system has been assembled.
- **interface:** A shared boundary.
- **module:** logically distinct part of a program
- **module or unit testing:** A series of tests performed on a program unit before it is integrated into a larger system.
- **needs analysis:** The process of studying user needs to arrive at a definition of system or software requirements.
- **preliminary design:** The process of analyzing design alternatives, defining the structure and relationships among the basic parts of the system, defining the interfaces, and typically preparing timing and sizing estimates.
- **portability:** The ease with which software can be transferred from one computer system or environment to another.
- **procedural specification:** The detailed description of a subroutine, a function, or a procedure.
- **production environment:** The conditions under which an application ultimately will operate.
- **proof of correctness:** An attempt to “prove” a program correct without running the program. These techniques can be considered a form of testing. Input and output assertions are formulated describing the program’s behavior; the goal is trying to prove that the program will conform to the output assertions from the given input assertions.
- **prototype:** A minimally functional system used to illustrate the software’s user interface or to prove the feasibility of a concept. *Caution:* This is a non-standard definition of the word prototype.
- **pseudocode:** A combination of programming language and natural language used for computer program design.
- **quality assurance:** A planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements.

- **real time:** Pertaining to the processing of data by a computer in connection with another process outside the computer according to time requirements imposed by the outside process.
- **requirement:** A condition or capability that must be met by a system or system component to satisfy a contract, specification, or other formally imposed document. The set of all requirements forms the basis for system development.
- **sandwich integration:** A system of integrating modules in a program that is a combination of top-down and bottom-up integration.
- **simulation testing:** Testing software with a simulation program and auxiliary hardware to imitate the "real" operating environment as closely as possible.
- **single-entry, single exit control structures:** Coding constructs that perform a well-defined task, have one entrance and one exit. Examples are IF-THEN-ELSE, DO-WHILE, REPEAT-UNTIL.
- **software:** Computer programs, procedures, rules, and associated documentation and data pertaining to the operation of a computer system.
- **software design description:** A document describing the major components of the software design including data bases and internal interfaces.
- **software development process:** The process by which user needs are translated into software requirements, software requirements are translated into design, the design is implemented in code, and the code is tested, documented, and certified for operational use.
- **software maintenance:** Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.
- **software professional:** One who develops or maintains software for others.
- **software reliability:** The ability of a program to perform a required function under stated conditions for a stated period of time.

- **software structure:** The overall representation of the software system showing information flow and structure determined from the requirements.
- **specification:** A concise statement of a set of requirements to be satisfied by a product, indicating, wherever appropriate, the procedure to determine whether the requirements given are satisfied. An example is a **system requirements specifications**.
- **structured design:** A disciplined approach to software design that adheres to a specified set of rules based on principles such as top-down design, stepwise refinement, and data flow analysis.
- **structured source code:** Computer programs derived from structured design.
- **subsystem testing:** checking interfaces between system parts
- **system:** An integrated whole composed of diverse, interacting specialized structures and subfunctions.
- **system testing:** The process of testing an integrated hardware and software system to verify the system meets its specified requirements.
- **testing:** Process of exercising a system (or some component) to identify differences between expected and actual results. [contrast with **debugging**.]
- **top-down integration:** A system of integrating modules in a program that starts with the top level module and adds subordinate modules. [see **bottom-up** and **sandwich integration**.]
- **validation:** The process of evaluating software at the *end* of the software development process to ensure compliance with software requirements. [see also **verification**]
- **verification:** The process of determining whether or not the products of a given *phase* of the software development cycle fulfill the requirements established during the previous phase. [see also **validation**]
- **walkthrough:** A workproduct review process where the author reads through the workproduct and a group of reviewers try to identify defects. [see also **inspection**]

## Appendix C

### Sample Sandia Module Header

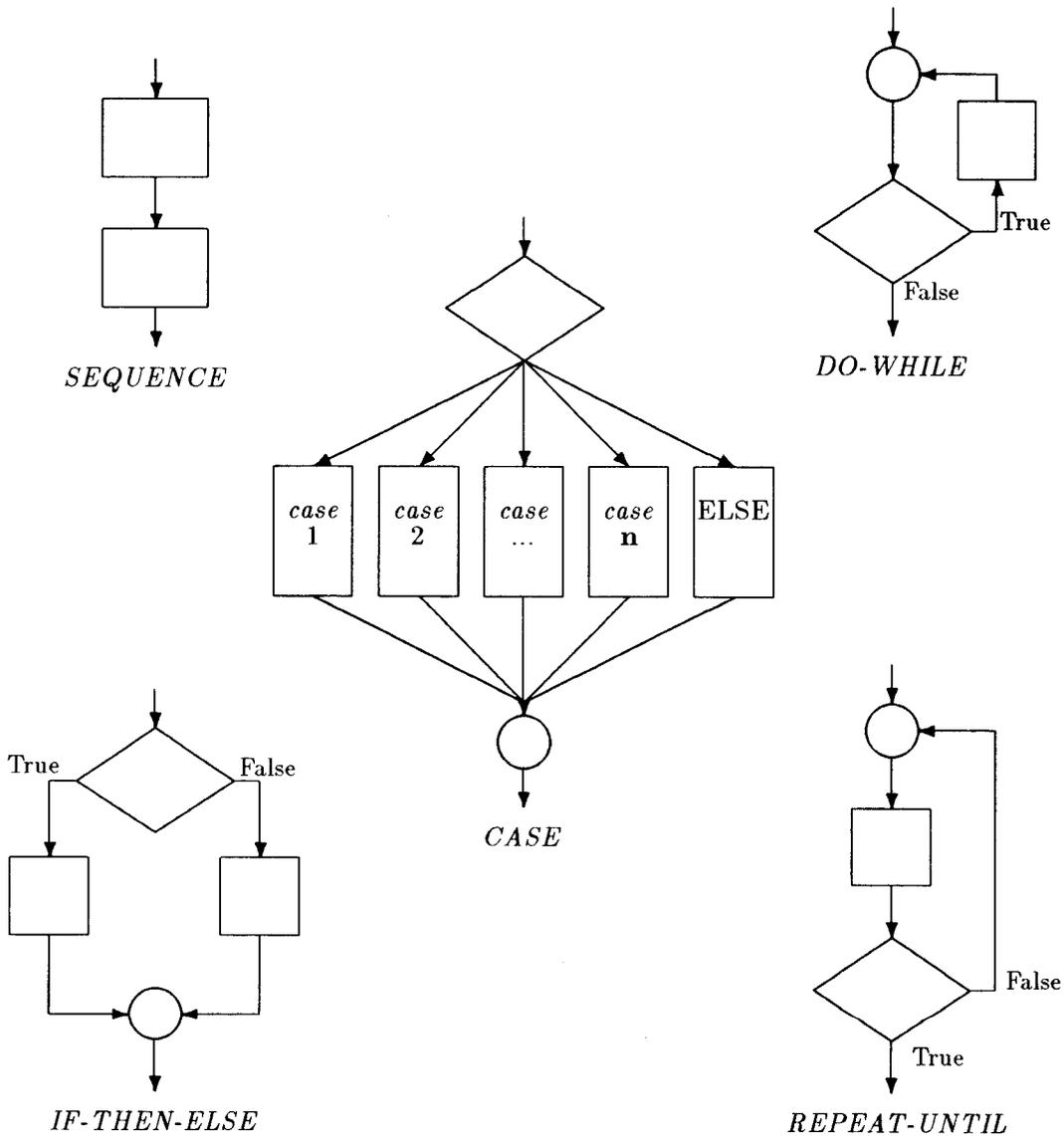
The following example provides a recommended module header format for Sandia software projects.

```
C=====
C          SUBROUTINE ENDPTS
C=====
C Description:          Create Coordinate File --
C          This module initiates the process to traverse the
C          drawing file and extract coordinate endpoint data,
C          either from a Vendor file or an IGES extracted file.
C
C          Programmer:   K.T. Bear, SNLA/2814, FTS 846-6014
C          Version:      1.100
C          Version Date: April 1, 1985
C          Environment:  DEC VAX 11/700/VMS 3.5, ANSI FORTRAN-77
C
C          module calls:   VENDOR, IGES
C          module called by: MENU01
C
C          inputs:         (drawing file name)
C          outputs:        (Coordinate Endpoint File)
C          assumptions:    Vendor file or IGES file has been opened.
C
C          last modified on: 02/22/86
C          last modified by: L. Marsupe, SNLA/2814
C          reason:         To increase local endpoints limit
C                          from 100 to 200 (ref: IGES 3.0.)
C
C          Local Variables:
C          LUIN           Vendor file or IGES file:      Integer
C          LUOUT          Coordinate endpoint file:       Integer
C          NPTS           Limiting number of endpoints:  Integer
C=====
```

## Appendix D

### Control Structures

The following graphics represent programming control structures advocated in section 5.2:



## Appendix E

### Inspection Report Forms

This Appendix provides example forms as developed by Sandia's Data Systems Division in response to requirements of the Software Inspection process, as described in section 9.

The forms include the following:

- **Inspection Profile**
  - To be completed by Author
  
- **Inspection Preparation Log**
  - To be completed by each Inspector
  
- **Inspection Management Report**
  - To be completed by Moderator
  
- **Inspection Defect List**
  - To be completed by Recorder
  
- **Inspection Summary**
  - To be completed by Moderator

# INSPECTION PROFILE

Project : \_\_\_\_\_

Date : \_\_\_\_\_

Unit : \_\_\_\_\_

Inspection Type :

Project Plan

Requirements

Design

Code

Test Plan

Test Cases

Installation Plan

Size of Material : \_\_\_\_\_ (unit) \_\_\_\_\_

Is this a reinspection :  No  Yes

Summary of Open Items : \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

Other Comments : \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_



# Inspection Management Report

Project \_\_\_\_\_ Unit \_\_\_\_\_ Moderator \_\_\_\_\_

Inspection type:

- Project Plan
- Design
- Requirements
- Code
- Test Plan
- Test Cases
- Installation Plan

Overview held:  No  Yes  
Overview Duration \_\_\_\_\_ Number attending \_\_\_\_\_

Number of inspection meetings \_\_\_\_\_ Total meeting duration \_\_\_\_\_  
Total number of inspectors \_\_\_\_\_ Total preparation time \_\_\_\_\_

Module disposition:  pass  follow-up  reinspect

Estimated rework effort \_\_\_\_\_ (days)  
Rework to be completed by \_\_\_\_\_  
Actual rework effort \_\_\_\_\_  
Reinspection scheduled for \_\_\_\_\_

Other inspectors

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Moderator certification \_\_\_\_\_ Date \_\_\_\_\_

Additional Comments

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_



## Index

Note: Page numbers in **boldface** reference a subsection on the indexed term. Page numbers in *italics* reference a definition of the term.

- acceptance testing **43**, 68, 78
- analysis phase 54
- ANSI FORTRAN-77 31
- assembler 29
  
- baseline 78
  - table 55
- bottom-up integration 78
- boundary condition 78
  
- change control **58**
  - board 59, 61
- change table 55, 57
- checklist 61
- code profilers 46
- cohesion 26, 78
- coupling 26, 78
- comments 35
- configuration
  - control 78
  - management 55
    - plan 55
- conversion **52**
  - direct, **53**
  - modular, **53**
  - parallel, **53**
- cpu 78
- critical design review 23, 79
  
- data
  - modular, **37**
  - organization, **37**
  
- debugging 68, 79
  - symbolic interactive, 47
- design, **15**
  - description 15, 18
  - detailed, 17, 23, 79
    - document 21
    - review 23
  - preliminary, 80
  - review **15**, 79
    - critical 23, 79
    - preliminary 79
    - results 15
  - structured, **24**, 82
- detailed design 79
- documentation,
  - in-line **35**
  - maintenance, 48
  - update supporting, 55
- documenting changes 57
- driver programs 46
- dynamic testing **40**
  
- egoless programming 67
- emergency changes 58, 59
- environment **1**
  - production 80
- error handling 32
  
- fixes,
  - emergency, 58
- formal reviews **62**
  
- HIPO 20, 24
  
- implementation **29**
- informal peer reviews **62**
- in-line documentation **35**

inspection **65**, 79  
 installation testing 79  
 integration 42, 44, 68, 79  
     librarian 56  
     of units 43  
     sandwich 81  
 integration testing 80  
     bottom-up, 78  
     top-down, 82  
  
 librarian 59  
     integration 56  
 library 38  
  
 maintenance 35, **54**, **61**  
     checklist 61  
     documentation 48  
     programmers 49  
     software 81  
 management,  
     configuration, 55  
     plan 55, 56  
 manual,  
     user's, 48  
 mnemonics 31  
 modular data 37  
 module 31, 80  
     header **36**  
     separation **36**  
  
 needs analysis 80  
  
 operational testing 79  
 operating procedures 48  
  
 pencil test **35**  
 plan,  
     configuration management 55,  
         56  
     project **5**  
         training, 48  
 portability **34**, 80  
 preliminary design 80  
     review 79  
 procedural specification 80  
 procedures,  
     operating **48**  
 processor dependent 34  
 production environment 80  
 project plan 5  
 prologue,  
     source code, 60  
 programming  
     egoless, 67  
     maintenance, 49  
 proof of correctness 80  
 prototype 54, 80  
 pseudocode 16, 20, 69, 80  
  
 quality assurance 80  
  
 real-time 44, 81  
     systems, testing of 45  
 requirements **10**, 81  
 review  
     critical design 23, 79  
     formal **62**  
     informal peer **62**  
     preliminary design 79  
     software requirements **9**  
 revision history 60  
 "rich" test set 41, 70  
  
 sandwich  
     integration 81  
     testing 42, 43  
 simulation testing 81  
 single entry, single exit 32  
 software 81  
     design description **18**, 81

- development process *81*
- life cycle **3**, *62*
- professional *81*
- reliability *81*
- requirements,
  - review **9**
  - specification **9**
- structure *82*
- test plan **39**
- source code prologue **60**, *60*
- spaghetti code **29**, *29*
- specification *82*
  - procedural *80*
  - software requirements **9**
  - system requirements *40*, *43*
- static testing *40*
- structured
  - design *82*
  - source code **29**, *82*
  - walkthrough *22*
- subsystem testing *82*
- symbolic
  - constants *34*, *47*
  - interactive debugger *47*
- system *82*
  - requirements specification *40*, *43*
  - testing *82*
- table,
  - baseline, *55*
  - change, *55*, *57*
- telephone test **35**, *35*
- test set *39*
  - documentation *39*
  - "rich" *41*, *70*
- test results *39*
- testing, **39**, *68*, *82*
  - acceptance **43**, *68*, *78*
  - dynamic **40**
  - operational *79*
  - installation *79*
  - integration *80*
  - real-time systems *45*
  - sandwich **42**, *43*
  - simulation *81*
  - static *40*
  - subsystem *82*
  - system *82*
  - top-down **42**
  - unit *68*, *80*
- top-down
  - testing **42**
- training plan *48*
- unit testing *68*, *80*
- update supporting documents *55*
- user's manual *48*
- validation **62**, *82*
- variable descriptors **37**
- verification **62**, *82*
- walkthrough **64**, *82*
  - structured *22*
- white space **29**, *29*

**Distribution:**

**Sandia Internal:**

333 R.D. Summers (2)  
341 P.S. Hamilton  
342 L.M. Ford (2)  
1231 P.L. McAllister (2)  
1254 T.F. Ezell (2)  
1500 W. Herrmann  
1520 D.J. McCloskey  
1523 J.H. Biffle (2)  
1533 M.E. Kipp  
1624 S.J. Weissman (2)  
1636 P.C. Kaestner  
2111 P. Hofstadler  
2113 J.A. Wisniewski (5)  
2113 J.A. Hudson (2)  
2300 J.L. Wirth  
2311 H.D. Pruett (2)  
2311 R.C. Lennox (2)  
2314 D.M. Small  
2330 E.H. Barsis  
2336 C.R. Borgman (2)  
2600 R.J. Detry  
2610 D.C. Jones  
2612 D.M. Darsey  
2614 A.R. Iacoletti  
2620 E.C. Domme  
2640 E.J. Theriot (2)  
2642 P.A. Lemke (2)  
2646 R.J. Hanson (2)  
2800 H.W. Schmitt  
2810 D.W. Doak  
2811 J.C. Kelly  
2811 L. Meirans (2)  
2812 J.F. Jones, Jr. (2)  
2812 L.M. Grady (5)  
2812 R.J. Harrison  
2813 S.K. Fletcher (2)  
2813 S.L.K. Rountree (5)

2813 D.B. Saylor (2)  
2814 P.A. Erickson  
2814 A.L. Ames (2)  
2814 M.A. Blackledge (100)  
2814 R.E. Parks (2)  
2820 G. Carli  
2821 R.E. Thompson  
2825 J.R. Yoder  
2826 A.J. Ahr (5)  
2830 G.R. Urish  
2850 J.L. Tischhauser  
2854 K.E. Wiegandt  
2854 S.C. Babb (5)  
2854 N.J. Nelson (2)  
2854 V.R. Yarberry (2)  
3151 R.L. Manhart  
4030 G.W. Kuswa  
5100 J.C. Crawford  
5146 J.W. Redel  
5164 D.H. Schroeder  
5164 M.W. Sharp  
5164 M.J. Smartt (2)  
5172 G.C. Novotny  
5172 G.J. Dodrill  
5218 D.J. Gould  
5255 P.W. Harris (2)  
5255 A.L. Yates (5)  
5261 D. Coleman  
5263 R.F. Davis (2)  
5268 C.E. Olson (2)  
5311 M.C. Jones  
5311 J.L. Krone  
5311 J.E. Lenberg  
5321 A.M. Maxted (2)  
5321 C.C. Newcom  
5321 D.H. Rountree (5)  
5324 L.J. Ellis (2)  
5324 M.T. McCornack (2)  
5324 J.C. Rowe (2)

5324 W.J. Slosarik (2)  
6228 P.J. Eicker  
6310 E.W. Shepherd (2)  
6312 R.W. Prindle  
6315 R.C. Hall  
6330 G.R. Romero  
6412 S.H. McAhren  
6415 F.E. Haskin (2)  
6415 L.T. Ritchie (2)  
6440 D. Brosseau (2)  
6440 D.A. Dahlgren (2)  
6444 J.M. McGlaun (5)  
7200 J.M. Wiesen  
7250 J.A. Hood  
7252 C.A. Trauth, Jr.  
7252 D.P. Patrick (5)  
7252 S.L. Sardalos (2)  
7262 F.A. Ross  
7262 D.G. Adams (5)  
7262 R.B. Ronan (2)  
7263 G.W. Mayes (2)  
7521 S.Y. Goldsmith  
7524 H.T. Cooley  
7524 W.D. Swartz (2)  
8025 R.L. Fugazzi  
8230 W.D. Wilson  
8235 D.L. Crawford (2)  
8270 R.C. Dougherty  
8272 D.B. Hall  
8274 R.J. Aiken  
8274 R.E. Isler (5)  
8274 P.K. Neighbors  
8348 T.P. Tooman (2)  
8474 J.N. Rogers (2)

3141 S.A. Landenberger (5)  
3151 W.L. Garner (3)  
3154-1 C. H. Dalin (28)  
for DOE/OSTI  
8024 P.W. Dean

Second Printing, October 1992

3827 Betty Straba (500)