STANDARDIZED DEVELOPMENT OF COMPUTER SOFTWARE PART II STANDARDS

Robert C. Tausworthe

STANDARDIZED DEVELOPMENT OF COMPUTER SOFTWARE

Robert C. Tausworthe

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

PART II
STANDARDS

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

Published in 1979 by Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

This material was prepared by the Jet Propulsion Laboratory under contract No. NAS7-100, National Aeronautics and Space Administration.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN: 0-13-842203-6

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA, PTY. LIMITED, *Sydney*
PRENTICE-HALL OF CANADA, PTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

PREFACE

At the time Part I of this work was being published, the Jet Propulsion Laboratory's Deep Space Network (DSN) was in the process of developing and writing a set of Software Standard Practices, for which Part I was cited as the "methodology textbook." The standards were developed over several years by a group we simply called the "Software Seminar." It was created and chaired in its "pathfinder period" by Walter K. Victor, who was, by the way, the significant inspirator of Part I. Mahlon F. Easterling then led that symposium (second Webster [43] meaning) through its next "pilot" phase. Edward C. Posner steered it through the arduous, major, final phase involving detailed standards development, consensus building, writing, review, and publication; he also sponsored the final symposium (first Webster [43] meaning) immediately after the standards were signed off by upper management. Daniel C. Preska administered the writing of the standards, with editorial assistance by Richard C. Chandlee.

The test-bed for the methodology reported in Part I had been an effort of medium magnitude—a program (the MBASIC™ language processor) containing about 25,000 lines of non-real-time assembly language code. The results of that methodology test-bed seemed to indicate that programmer performance better than had been encountered in past DSN projects could be extended to the DSN as a whole—an organization involving perhaps 100 programmers in various disciplines.

With that belief, the implementation team manager of a critical hardware/software project—to completely upgrade the digital data systems in all of the deep-space stations around the world—undertook the additional task of applying and evaluating the then-emerging DSN Software Standard Practices as a standards-test-bed activity. The overall project, including software for system performance tests, generated approximately 100,000 lines of hard-real-time assembly language code over about 2-1/2 years.

That project could ill-afford to be a mere guinea pig for a software standards seminar, because the delivery of the first-phase system was crucially tied to upcoming spacecraft launch dates and committed on-going

missions. Even moderate deviations from the original schedule could not be tolerated. Short slippages could, perhaps, be accommodated if detected early enough for appropriate replanning.

Yet the prospects for success using the standards seemed good. A software manager was appointed, cognizant software development engineers for each of the major assemblies making up the system were selected, and a secretariat function (Chapter 17) was established. Subcontractors were selected to aid in all activities of the implementation and integrated with JPL personnel into a unified team. All were admonished to apply and conform to the (draft-form) software standards to the maximum extent, except where it could be shown that adherence to standards was interfering with the schedule. Waivers were granted on a case-by-case basis, in writing, to record the details wherein standards proved ineffective.

The project demonstrated numerous gratifying benefits arising from the methodology presented in Part I and the more detailed standards contained in this second volume. Among these were good schedule and cost performance, high product reliability, adequate documentation, increased productivity, and smooth development and delivery.

The delivery date did slip from the original 2-1/2 year plan by somewhat less than 1 month (3% accuracy of original plan). However, this slip was predicted about 7 months in advance of its actual occurrence, so that effective contingency planning could be initiated. The 6% cost overrun was also predicted well enough in advance that reallocation of project resources was effective. These excesses were considered unusually slight, particularly in comparison to past JPL experience and then-current industry-published data.

The software contained an average of approximately 3 errors per thousand lines of code, measured from the beginning of system integration tests, as compared to 10–20 errors per thousand lines commonly reported in similar projects not employing top-down structured programming methodology. This test phase, as a matter of fact, required only about 15% of the overall effort, whereas industry-published figures and previous JPL experience quoted about a 50% level of effort. The difference in effort was expended in the design and planning phases to produce a more mature, well-documented, reliable product.

The development and delivery were reported as being smooth and controlled. No "tiger teams" were required during implementation, no significant renegotiation of software commitments was needed near the end

of production, and the software was delivered ready for operation with very few liens levied for future corrective action.

The standards, of course, did not accomplish these achievements—*people* did. JPL was fortunate to have had outstanding personnel performing in an exceptional, professional manner throughout the project. All that one may claim for the standards is that they provided a methodology which allowed each member of the project to apply himself or herself toward the accomplishment of project goals in the most effective way.

That methodology held up to its promise. The managers, designers, coders, operational personnel, documentarians, and theoreticians in concert had crafted and codified a viable, detailed set of practices for producing software. All concerned had had a voice in the creation and adjustment of their software engineering discipline, and for once the "horse" designed and built by a "committee" didn't turn out to be a "camel."

Part II of this monograph, then, exposes this detailed set of rules for software implementation. I have broadened some of the DSN practices in some instances, in an attempt to make them more readily adaptable to organizational structures different than that of the DSN. Additional consonant practices from other sources have also been incorporated to broaden the scope of applicability to projects of types other than the high-technology, high-efficiency, single-purpose, custom-built variety demanded by the deep-space-station environment.

There are many whom I must thank and acknowledge for their many and various contributions toward the completion of this second volume. Robertson Stevens, the former manager of a large computing facility and, during this time, manager of the upper-level organization containing the hardware/software project, was the propounder of many of the management policies and status monitors that are found in this work. Paul T. Westmoreland was the manager of the implementation project; his professionalism, ability to manage, faith in a standardized approach, and courage to commit that approach to a critical task have been a personal inspiration.

I must also acknowledge the effectiveness of Alvin F. Ellman of the Bendix Corporation, who was software manager. It was perhaps Al's ability to recognize what quantitative information a programmer could communicate naturally to management and others that led to the refined status monitors that proved so effective. His ability to relate to and interface with project-internal programmers and project-external systems engineers and users was a major factor in an organization-wide feeling of confidence in the health of the growing and maturing software.

The subsystem Cognizant Development Engineers were Robert Desens, Frank Hlavaty, Ronald Murray, Gary Osborn, and Steve Yee. Observance of their applications of the standards and their performance under the standards produced many refinements for effectiveness.

The members of the DSN Programming System Steering Committee included, at various times, Walter K. Victor, Robertson Stevens, Mahlon Easterling, Lee W. Randolph, Carl W. Johnson, Cecil P. Wiggins, Edward C. Posner, Malvin L. Yeater, William C. Frey, William D. Hodgson, Angela Irvine, Raul D. Rey, Richard B. Miller, and Donald L. Gordon. Each made special contributions too numerous to single out.

R. Booth Hartley and Lawrence R. Hawley were both co-developers and appliers of the rules given here during the various implementations of elements of the DSN Programming System. Their support, feedback, and ability were sorely needed and freely provided during the preparation of this material. Kay Landon and Leonard Benson proofread Part I and generated its index; they also progammed a prototype CRISPFLOW processor, leading to the descriptions in Appendix G. Annamarie Grana helped evaluate the utility of Appendix C, using it as a guide for the generation of two SRDs. Frank Hlavaty collaborated in the formation of Appendix E. Michelle Martin and Marshall Polsley contributed to the format and content of Appendix I. Richard Schwartz's influence is prevalent in the standard language discussions in Chapter 17. John Johnson and Henry Kleine were instrumental in the formation of the CRISP language.

I give special thanks to Georgiana Clark, who typed and corrected the entire manuscript; to Carol Rosner, who had typed a preliminary draft of the first five chapters; to Margaret Seymour, who drafted all the figures except those in Appendices G and L; to Silvia Munoz, who aided in generating the index; to Harold Yamamoto, who edited the volume for publication; and to Doris Perry, who coordinated all the artwork and was responsible for the final page makeup. I also extend a belated thanks to Anita Sohus, who coordinated all the artwork for Part I.

Finally, I wish to thank those who have participated in the many seminars and classes given on this material during its various stages of completion; many insights into the secrets of software engineering across a broad programmer base were revealed to me as the result of these interactions.

Robert C. Tausworthe

Part II
STANDARDS

CONTENTS

PART II

PART I

XI. SOFTWARE REQUIREMENTS AND DEFINITION STANDARDS

This chapter is the first of a set containing specific standards extracted from, or generated in response to, the methods presented in Part I. These are the rules that guide the top-down, hierarchic, modular, structured approach to software development.

There are, of course, no universal rules to make intricate programming a simple task, and there is perhaps very little hope of ever completely formalizing the programming process. Design is a creative, inventive craft. But merely identifying the constraints, objectives, design tools, and parameters in a standardized way yields considerable progress in dealing with problems effectively. Furthermore, these standard procedures can be taught. References [1] through [6] are examples of standards in effect based on the methodology reported here.

The standards and practices contained in the remainder of this work are meant primarily to apply to new programs or major extensions to existing programs intended for operational use. They are meant to be easy to use, to

be somewhat flexible, and to provide guidelines for focusing the activities toward what is most needed.

The use of a consistent outline and format for documenting each activity is presumed. The outlines in Appendices C, D, and E contain a detailed set of topics to be considered in defining the requirements and functional behavior of a software package. The topics also give guidelines as to what material is to be specified within each topic.

A large portion of any software engineering activity deals fundamentally with the planning of a software development, rather than the actual doing of it. I recognize that a discipline for such planning is needed just as much as a discipline for doing, so I have oriented the rules given here toward the more technical aspects of project engineering and software development. The interested reader wishing to delve more deeply into management and planning disciplines may consult [1] through [11].

11.1 GENERATING SOFTWARE REQUIREMENTS

A software requirement is a need established for a piece of software by an organization in order to achieve certain goals. The requirement-generation activity culminates in the approvals, negotiations, and commitments of resources necessary to initiate, sustain, and complete the software development. Although I have not considered requirements generation in past chapters to be among the software development activities, nevertheless, there are a few guidelines that can make the generation of software requirements more uniformly responsive to the needs of oncoming activities.

The Software Requirements Document (SRD) is, relatively speaking, a non-technical document; in its first-reviewed form, it probably contains only enough functional and technical information and requirements (perhaps by reference) to identify the need for a perhaps intangible capability. At this point, its content is primarily oriented so as to allow the authorizing organization to determine what it is approving.

Part of this approval involves the expenditure of resources to permit the requirers (and, later, implementors) to supply more planning information and technical detail.

Requirements are only definite to the extent that they are documented. The needed output of the requirement-generation activity is an SRD satisfying the following criteria (see Section 3.3):

a. It should be adequate to identify the objectives of the program, its environment, the configuration needed for its operation, the resources required for its support, and the advantages and disadvantages in the service it provides, as related to the customer organization.

b. It should be adequate to permit the developmental activities to proceed under a reasonable assurance that major revision of requirements will not be necessary.

c. It should be adequate for review and approval by cognizant authority on the basis of its conceptual feasibility in accordance with the other criteria above. It should contain manpower, schedules, and development-cost estimates, as well as reasonable variances for these estimates, at least for the next phase of activity.

As I indicated in previous chapters, it may not always be feasible to actually complete the SRD until after some of the software development process has already begun (in a top-down way, of course). That part upon which the funding and manpower approvals are based (the overview) probably derives in largest part from the justification section (see Appendix C). This justification—intended for management—contains material oriented principally toward establishing the need for, and feasibility of, software to fulfill certain goals or missions of the funding organization.

The remainder of the SRD—for guiding the implementation—deals with setting forth technical requirements, developmental constraints, and acceptance criteria in enough detail to identify the external functional and operational characteristics of the software. These can subsequently be negotiated, refined, and then implemented so as to satisfy the sponsor's goals. The final SRD constitutes an agreement between the requesting and implementing organizations on the software to be produced.

The SRD contains material that may well be broken into several separate documents, such as, perhaps, a Software Justification Report, a Software Acquisition Plan, a Software Functional Requirement, and so on. Some material may be included directly, if not extensively, while some may be appended or referenced. The hierarchic nature of the outline in Appendix C permits this to be done quite easily in the most accommodating way, should the need arise.

The reader may notice, comparing Appendices C, D, and E, that the SRD, Software Design Definition (SDD), and Software Specification Document (SSD) outlines are all very similar in appearance. But while they cover the same general topic, they do not generate the same content. The SRD, upon completion, contains customer/user requirements and constraints, and estimates the resources available or needed for software

production. The SDD identifies those external characteristics of a program needed to fulfill the given requirements, establishes the program architecture, defines costs and schedules, and presents a work breakdown structure by tasks. The SSD defines the external and operational characteristics of the program and specifies how these are to be effected into internal program structures, "as built." By giving these documents the semblance of a conformable outline, I have tried to ensure a rough downward traceability from pre-design requirement to definitions, to implemented specifications, and upward, again, in the reverse order.

11.1.1 Rules for Generating Software Requirements

The following guidelines for structuring software requirements form a small set of standards to aid in the preparation of the SRD through the overview phase, culminating in a Requirements Review. Sections 3.1 and 3.2 contain useful guidelines for recognizing requirements.

1. Complete the SRD, supplying the material indicated in the topical outline (Appendix C), documented subject to the rules in Section 11.4.

> Delete items in the outline that are irrelevant or do not apply, or mark them as "not applicable." Mark purposely unspecified items as "development prerogative," perhaps conditioned by the addition of modifiers, such as "subject to approval." Cite existing material, such as policies and constraints, by reference; include deviations from these, however. Figure 11-1 presents a graphic table of contents for the SRD that emphasizes the hierarchy of management information. The complete graphic table of contents is shown as Figure C-1 (Appendix C).

2. Establish a set of criteria and weights for the SRD review, such as breadth of coverage, level of detail, adherence to standards, etc., and obtain concurrence on these from the review board.

3. Identify cost and schedule drivers likely to impact the decisions that have to be made. Establish priorities and constraints for development, with emphasis placed on factors that tend to drive costs up if they are not identified and frozen early.

> Include any known factors involving special complexity that tend to jeopardize schedules or place stringent demands on specific (and possibly scarce or unavailable) personnel.

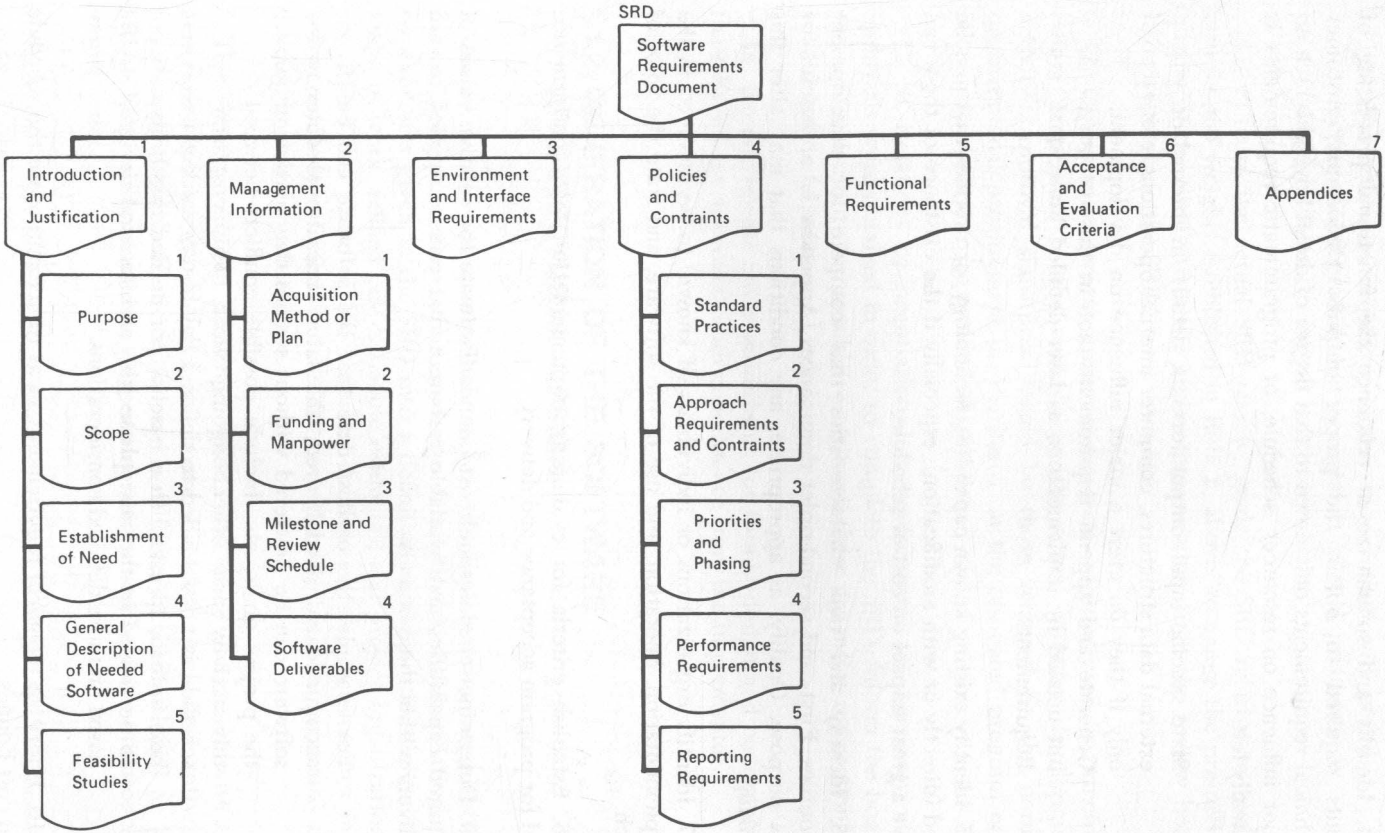

Figure 11-1. A visual table of contents for the SRD, showing hierarchy of management items important for approval phase