Preventing software requirements specification errors with IEEE 830

AFTER a software development schedule has been established, we often find it does not include enough time to complete all necessary work. At that point we begin to make compromises. In making the compromises we usually throw out everything in favor of the schedule. Under pressure from a schedule, managers will often issue hasty directions. "Don't document now. We'll document later." "We don't have time for a review. Just go with what you've got." "We don't have time to learn a standard technique or a new tool on this project."

In a future article we will look at why software development schedules are often unrealistic in the first place. But for now, we want to look at ways to come closer to meeting the schedules to which we are already committed by using IEEE 830 and some simple rules. The panic compromises of documenting later, skipping reviews, and using nonstandard techniques or tools will usually shift errors from the immediate phase into the testing phase where they are more costly to correct.

**Errors in requirements.** A typical software development cycle is illustrated in Figure 1. We can see from this illustration that over 40 percent of the software development effort is spent in testing. If we are trying to meet a tight schedule, we want to reduce the time spent in testing, which includes failure identification, fault isolation, and fault removal—the biggest piece of the pie in Figure 1. (If your schedule does not include 40 percent or more of the total development time for testing, your effort is significantly different from currently reported norms.)

Another factor pushing us toward reducing test time is the cost to eliminate an error in the different life-cycle phases as shown in Figure 2. We see it is much less expensive to eliminate errors early. The errors that give us the most trouble in testing usually originate in the requirements and design phases, so they are there early on for us to find as shown in Figure 3.

An ounce of prevention is worth a pound of cure. A number of well-known authors have presented various techniques for finding software errors early in the development cycle. Finding errors early is obviously advantageous, but preventing errors from occurring in the first place is obviously the best course of action.

![Figure 1. Software development cost.](image)

![Figure 2. Fault removal cost.](image)

![Figure 3. Fault removal cost (derived from Alberts 1976).](image)
Characteristics of a good SRS. A good SRS is

1. unambiguous,
2. complete,
3. verifiable,
4. consistent,
5. modifiable,
6. traceable, and
7. usable during the operation and maintenance phase.

Unambiguous. An SRS is unambiguous if—and only if—every requirement stated therein has only one interpretation.

1. As a minimum, this requires that each characteristic of the final product be described using a single unique term.
2. In cases where a term used in a particular context could have multiple meanings, the term must be included in a glossary where its meaning is made more specific.

Complete. An SRS is complete if it possesses the following qualities:

1. Inclusion of all significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces.
2. Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to valid and invalid input values.
3. Conformity to any SRS standard that applies to it. If a particular section of the standard is not applicable, the SRS should include the section number and an explanation of why it is not applicable.
4. Full labeling and referencing of all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.

Use of TBDS. Any SRS that uses the phrase, "to be determined," is not a complete SRS.

Verifiable. An SRS is verifiable if and only if every requirement stated therein is verifiable. A requirement is verifiable if and only if there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement.

1. Examples of nonverifiable requirements include statements such as "The product should work well," or "The product should have a good human interface." These requirements cannot be verified because it is impossible to define the terms good or well.
2. An example of a verifiable statement is, "The output of the program shall be given within 20 seconds of event X, 60 percent of the time; and shall be given within 30 seconds of event X, 100 percent of the time." This statement can be verified because it uses concrete terms and measurable quantities.
3. If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement should be removed or revised.
4. If a requirement is not expressible in verifiable terms at the time the SRS is prepared, then a point in the development cycle (review, test plan issue, etc.) should be identified at which the requirement must be put into a verifiable form.

Consistent. An SRS is consistent if and only if no set of individual requirements described in it conflict. There are three types of likely conflicts in an SRS.

1. Two or more requirements might describe the same real world object but use different terms for that object. For example, a program's request for a user input might be called a prompt in one requirement and a cue in another.
2. The specified characteristics of real world objects might conflict. For example, the format of an output report might be described in one requirement as tabular but in another as textual.
3. There might be a logical or temporal conflict between two specified actions. For example,
   (a) One requirement might specify that the program will add two inputs and another specify that the program will multiply them; or
   (b) One requirement might state that A must always follow B, while another requires that A and B occur simultaneously.

Modifiable. An SRS is modifiable if its structure and style are such that any necessary changes to the requirements can be made easily, completely, and consistently. Modifiability generally requires an SRS to

1. have a coherent and easy-to-use organization, with a table of contents, an index, and explicit cross-referencing; and
2. not be redundant; that is, the same requirement should not appear in more than one place in the SRS. Redundancy itself is not an error, but it can easily lead to errors.

Traceable. An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. Two types of traceability are recommended:

1. Backward traceability (that is, to previous stages of development) depends upon each requirement explicitly referencing its source in previous documents.
2. Forward traceability (that is, to all documents spawned by the SRS) depends upon each requirement in the SRS having a unique name or reference number.

Usable during the operation and maintenance phase. The SRS must address the needs of the operation and maintenance phase, including the eventual replacement of the software.

Most often, the need for a new software product is identified by a customer or a marketing representative. When this happens, the development cycle and the requirements phase are kicked off.

The first step in the requirements phase is to analyze the need pointed out by the customer or marketer. This activity may be called problem analysis, system analysis, or needs assessment. Whatever it is called, it amounts to a thorough examination of the need or problem.

The second step in the requirements phase is to look at ways to fulfill the need or to solve the problem. This activity often is called a feasibility study, and it identifies

Figure 4. Characteristics of a good SRS derived from IEEE 830.
the selected method of satisfying the need or solving the problem.

If the feasibility study indicates that a software product is the best answer, a software requirements specification will be generated. A requirements specification is a detailed description of what a product must do. In this final step of the requirements phase, IEEE 830 comes into play.

IEEE 830 is a description of what makes a good software requirements specification. It offers the format, content, and qualities of an understandable specification. By employing 830, a software development team can prevent errors in at least two ways.

Misinterpretations among developers are much less likely to occur when 830 is being used, because developers will be adhering to standard definitions of terms in writing requirements and plugging information into a prescribed format.

IEEE 830 also provides a list of the characteristics of a good software requirements specification. The reciprocals of the characteristics on the list may be considered most probable errors. If a developer knows the most probable errors that occur in a specification, he can avoid making them in the first place. The characteristics on the list and their definitions are shown in Figure 4. Let's look closely at one characteristic.

Listed as a desirable characteristic is an unambiguous requirement. The most probable error, then, is an ambiguous requirement. An ambiguous requirement is one that can be interpreted in more than one way. For example, "The character string will be terminated by a delimiter." The reader of such a requirement does not know if the delimiter is the last character in the string, or a simple count of the characters in the string, or if some special character will be designated the delimiter. IEEE 830 provides formal definitions of the characteristics and examples of bad requirements or most probable errors as well as examples of good requirements. IEEE 830 steers us clear of poor workmanship and guides us toward good professional practice.

There are holes in 830. For example, in providing prototype outlines, 830 does not specify explicitly how to fill in the sections. The prototype outlines shown in Figure 5 indicate what paragraph should be present but do not tell us how to specify the information that goes into the paragraphs. So we

---

### Table of Contents

1. Introduction
   1.1 Purpose
   1.2 Scope
   1.3 Definitions, Acronyms, and Abbreviations
   1.4 References
   1.5 Overview

2. General Description
   2.1 Product Perspective
   2.2 Product Functions
   2.3 User Characteristics
   2.4 General Constraints
   2.5 Assumptions and Dependencies

3. Specific Requirements

3.1 Functional Requirements
   3.1.1 Functional Requirement 1
   3.1.1.1 Specification
   3.1.1.2 Inputs
   3.1.1.3 Processing
   3.1.1.4 Outputs
   3.1.1.5 External Interfaces
   ... 

3.2 Performance Requirements
   3.2.1 Standards Compliance
   3.2.2 Hardware Limitations
   ... 

3.3 Attributes
   3.3.1 Security
   3.3.2 Maintainability
   ... 

3.4 Other Requirements
   3.4.1 Database
   3.4.2 Operations
   3.4.3 Site Adaptation
   ... 

4. Appendixes
5. Index

---

Figure 5. Prototype SRS outlines.
Rule 1. Use IEEE 830 for
  - a standard format,
  - a completeness check, and
  - a checklist of good requirement characteristics.

Reason: To establish consistency between documents.

Rule 2. Use models for
  - functional relationships,
  - data flow,
  - data structure, and
  - performance.

Reason: To express complete requirements.

Rule 3. Limit the structure of paragraphs to a list of individual sentences.

Reason: To increase the traceability and modifiability of each requirement; and to increase the ability to check for completeness.

Rule 4. Limit the structure of each sentence to a simple sentence that is, noncompound verbs or objects.

Reason: To increase the verifiability (testability) of each requirement.

Rule 5. Limit the verbs and objects in the sentences to a small set with a single specified definition for each word.

Rule 6. Limit the verbs and objects to terms that are common to the end user of the product (that is, nonprogrammer terms).

Reason: To increase user understanding of the requirements.

Rule 7. Limit the verbs and objects to actions and items that are visible external to the product.

Reason: To reduce the amount of design data that goes into the requirements; and to increase the testability of the product.

Figure 6. Rules of order for specifying software requirements.

see that 830 is not a panacea, but it does take a significant step toward establishing a reference for professional practice, and it can help us prevent errors.

Some rules to help us specify the necessary information in accordance with the 830 list of good requirements characteristics are provided in Figure 6. These rules are important. They do for requirements specifications exactly what the structured programming constructs do for programming. They reduce the probability of errors.

In the United States, at least four organizations are presently applying IEEE 830. To date, we at the Standards Department have received mostly favorable comments about these efforts. We are now actively seeking “war stories” dealing with IEEE 830 as well as with all other IEEE standards. Please contact Bob Poston if you have any experiences to share.