Requirements writing know-how
Robert M. Poston,
Software Standards Editor

How do I write good requirements for software products? This question echoes across our industry over and over again.

The IEEE began to address the challenge of writing “good” requirements specifications back in 1979 and finalized a Guide to Software Requirements Specifications (Std 830) in 1984. This guide has been a big help to software engineers. I see it being used more than any other software engineering standard. But it is not without its problems.

Std 830 was developed by hundreds of professionals across the country and overseas. That means it represents a high-level, international view of what software requirements specifications should be. The concepts and document outlines included in the standard are necessarily generic and nonspecific.

Getting from the high level to the specific is a problem for most people who use Std 830. The good news is that, through constant use, techniques have evolved to make using Std 830 easier.

Perhaps some of them can help you.

Homework first. Before we begin to write requirements for any project, we should do some preparatory research. Will the end user want the product? Will the end user save or make money with the product? Will we make money on it? Can we implement the product with technologies we know today? Is the product being founded on a technology that will still be timely when the product is ready?

Many times, an analyst will specify requirements (or a programmer will code a product) that he thinks the user needs. The user is not consulted about what he feels he needs. All concerned suffer a big disappointment when the user rejects the product.

An up-front needs/problems analysis might have prevented such a let-down. The moral here is to get the user involved as early as possible.

Probably the most fundamental question a user (buyer) will ask is, “What will I get out of this product?” The user wants some kind of return on his investment. Makers of the product have similar concerns.

Economic analysis helps make the go/no-go decision. This gets the business manager into the picture.

There is little good in specifying a terrific product that can’t be successfully implemented. Technologies can be pushed only so far.

For example, a relational database under consideration has proved flexible and easy to use, but its response times are slow and unpredictable. Will the response-time problem get in the way of our new product?

We always need a feasibility study in the beginning. Designers and testers are brought in here.

Several companies opted a few years ago to build their own computer boards for inclusion in their products. By the time these companies produced their boards, mass-market competitors had latched onto new technologies and were manufacturing more powerful boards at a fraction of the in-house cost. Building their own boards quickly became impractical for these companies.

Long-range (two years in our industry) technology forecasting must be done. Enter all those with crystal balls.

Often, all these activities are lumped together under one title: concept exploration. By doing this preparatory work, we significantly increase our chances of specifying a successful product.

Getting started. There are four prototype outlines in Std 830 dealing with specific requirements. Each covers functional, external interface, performance, design constraints, and quality (attribute) requirements. Each outline shows a slightly different partitioning of the requirements information.

Which one is best for your project? Simply choose the one that is partitioned most conveniently for the readers of your requirements document.

I have entered the prototype outlines into my word-processing system. On a new project, I experiment with the outlines. I try to reduce the number of cross references among the sections in the document.

Anytime I have to state something more than once or reference previously stated information, I get worried. The threads of information become too tangled for me. This tells me that I probably need to reorganize. When I have grouped my information correctly, a change in a requirement will affect only one section in my outline.

Data definitions and functional descriptions in the user’s world and in the product specification should correspond. This makes for minimum crossover in outline sections. (It will also help localize changes in the design of the final product.)

To help myself limit and focus on the particular information that belongs in one section, I constantly think of the purpose of each section (see the box on p. 71). I store my information in separate software files so I can move information around in my outline with simple editing commands.

With outline in hand, the actual requirements writing can begin.

Detailing information. It’s easy to move through section 1 of the overview prototype outline with Std 830. But, what belongs in section 2? Section 2 is titled “General Description,” and it is too general for most people to use. So let’s get down to some troublesome areas in that section.

Know the user. Subsection 2.3 is called “User Characteristics.” Here, we’re supposed to describe the educa-
tional level, experience, and technical expertise of people who will use the product. At first that seems straightforward, but what do you write when the user is a robot?

Many times, subsection 2.3, as well as other sections, may not be applicable to certain projects. Just leave the section number and title, and write "not applicable" in that space. That indicates the requirements writer considered the issues in the section and decided the section was not needed for this product. This is different from deleting or omitting sections. Anyone who must go back to an old requirements spec will appreciate the value of the "not applicable" label.

How much use? Frequency of use is a very important user characteristic. Yet, it is often omitted from requirements specifications. Std 830 is a little vague about frequency of use.

A user who works on a system for an hour a week probably will prefer menu-driven rather than command-driven products. It's a pain to memorize commands for occasional use. But, what about the person who pounds the keys for many hours a day? A menu will only get in his way.

So, identify in subsection 2.3 all end users and, most importantly, document how often each one will use the product.

Define, don't design. Now let's look at the heart of the software requirements specification, section 3, titled "Specific Requirements."

Std 830 states, "The SRS should not normally specify design items such as "(a) partitioning the software into modules "(b) allocating functions to the modules "(c) describing the flow of information or control between modules, and "(d) choosing data structures.""

Indeed, most experts in requirements writing agree that design information in a requirements spec restricts later design. It clutters the specification with irrelevant information.

In spite of this advice, nearly every requirements spec I see tells me about subsystems or modules and sequences of operations. How can we write cleaner requirements specs that are free of design information?

Data in any requirements spec has to be defined with nouns. Nouns that name things inside a product like files, registers, buffers, etc. are concerned with design. These nouns must be avoided in a requirements spec.

Stated another way, the requirements spec may contain data descriptions for

### Prototype outline

**Dwayne Knirk, Programming Environments, Inc.**

This prototype software requirements specification outline will help remind you of the purpose of each section.

<table>
<thead>
<tr>
<th>1. Introduction</th>
<th>1.1 Purpose</th>
<th>1.2 Scope</th>
<th>1.3 Terminology</th>
<th>1.4 References</th>
<th>1.5 Overview</th>
<th>1.5.1 Spec organization</th>
<th>1.5.2 Spec audience</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Describe the intent of the document.</td>
<td>Define the extent of material included.</td>
<td>Define application-specific terms used.</td>
<td>Identify related material.</td>
<td>Help reader locate information.</td>
<td>Define for whom the document is produced (buyer, definer, builder, and verifier).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2. General description</th>
<th>2.1 Product perspective</th>
<th>2.2 Product functions</th>
<th>2.3 User characteristics</th>
<th>2.4 General constraints</th>
<th>2.5 Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Establish a context for the product (a high-level management overview).</td>
<td>Describe the physical product aspects.</td>
<td>Describe the functional product aspects.</td>
<td>Describe the intended users.</td>
<td>Describe sources of limitations.</td>
</tr>
<tr>
<td></td>
<td>Define policies for error prevention, detection, notification, and correction.</td>
<td>Define how the product interfaces through its external boundaries.</td>
<td>Characterize interfaces with people.</td>
<td>Characterize interfaces with hardware.</td>
<td>Characterize interfaces with software.</td>
</tr>
<tr>
<td></td>
<td>Define how the product interfaces with physical environment.</td>
<td>Define acceptability of operation.</td>
<td>Define maximum allowed duration of outage.</td>
<td>Define maximum frequency of failure.</td>
<td>Define protection for processes and data.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. Specific requirements</th>
<th>3.1 Functional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 External interface requirements</td>
<td>Define dataflow volume and rates.</td>
</tr>
<tr>
<td>3.2.1 Human</td>
<td>Define retained data volumes.</td>
</tr>
<tr>
<td>3.2.2 Hardware</td>
<td>Define stimulus-to-response times.</td>
</tr>
<tr>
<td>3.2.3 Software</td>
<td>Define operating costs (hardware limits).</td>
</tr>
<tr>
<td>3.2.4 Communications</td>
<td>Identify limitations on design decisions.</td>
</tr>
<tr>
<td>3.2.5 Location</td>
<td>Identify mandatory system behavior.</td>
</tr>
<tr>
<td>3.3 Performance Requirements</td>
<td>Specify hardware capability bounds.</td>
</tr>
<tr>
<td>3.3.1 Input/output loads</td>
<td>Define acceptability of operation.</td>
</tr>
<tr>
<td>3.3.2 Database loads</td>
<td>Define maximum allowed duration of outage.</td>
</tr>
<tr>
<td>3.3.3 Response times</td>
<td>Define maximum frequency of failure.</td>
</tr>
<tr>
<td>3.3.4 Resource usage</td>
<td>Define maximum allowed time to repair.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3.5 Attributes (quality requirements)</th>
<th>3.5.2 Reliability</th>
<th>3.5.3 Maintainability</th>
<th>3.5.4 Security</th>
<th>3.5.5 Usability</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1 Availability</td>
<td>Define protection for processes and data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6 Other requirements</td>
<td>Define user suitability (effort or time to learn and effort or time to use).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.1 Database</td>
<td>Define the database (not the database system).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.2 Operations</td>
<td>Define support operations (startup, backup, etc.)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.3 Site adaptation</td>
<td>Define operations or data that change for each site.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
only that data visible outside the software to be built. The same idea applies to using verbs in a requirements spec.

Another way to keep design information out is to put the functional statements into one unordered list (ordering is the first step in design). If there is one list, no partitioning to subsystems can occur (the second step in design).

Unambiguous functional requirements. "The user changed his mind again," "That's not what I thought the requirement meant." We hear such frustrations being vented on most software projects. A lot of these misunderstandings are rooted in ambiguous functional requirements set forth in section 3 of the requirements spec.

An unambiguous statement can have one, and only one, interpretation. Very well! But, how do we develop such a fool-proof statement in English?

Programmer-to-programmer communication is difficult enough. How can we get our ideas across to others who will review and approve the requirements spec: the end user who happens to be an attorney, the tester who is a hardware engineer, the project manager who is a mathematician, and so on?

Every requirements spec must have its own data and verb dictionaries appended. Each term in these dictionaries must be assigned one, and only one, meaning. All people involved with the requirements spec must abide by those specific definitions.

Graphics representations of functional statements cannot minimize ambiguities if words on the charts and pictures are not strictly defined. There is no way around using controlled dictionaries.

A good job? At the end of the writing effort, we need to know if we've done a good job on the requirements spec.

Often a nod from the boss is all the approval we seek. Unfortunately, most managers are too busy to thoroughly evaluate requirements specs. We need our own way to evaluate our requirements spec before we release it to any one. Here are some simple definitions to go by:

A good requirement statement is a description of a required function or characteristic that is

- uniquely identifiable (traceable),
- externally observable (testable and does not restrict design), and
- unambiguous (with one, and only one, interpretation).

A good requirements specification is a set of requirements statements that are

- complete (as agreed to by the builder, buyer, and verifier) and
- consistent (with no internally conflicting definitions or concepts).

Mail order Std 830. Copies of ANSI/IEEE Std 830-1984, Guide to Software Requirements Specifications (order number 933) can be ordered from Computer Society Press Order Dept., PO Box 80452, World Way Postal Center, Los Angeles, CA 90080. The single copy price for IEEE members is $6.30 ($7 nonmembers). Please add $2 for shipping and handling.

STANDARDS BULLETIN

X/Open endorses IEEE Posix

X/Open, an international software portability initiative by 11 major information suppliers, has endorsed the IEEE proposal to establish Posix, a portable system interface definition, as a formal industry standard.

X/Open has historically based its standard support on AT&T's Unix System V interface definition, the de facto industry standard. However, the broad market support for the IEEE standard (including the US National Bureau of Standards) led X/Open to accept Posix as a formal industry standard.

To help further the Posix effort, X/Open established a Posix working group of its own and assigned two technical staff members to the project.

The X/Open members are AT&T, Bull, Digital Equipment Corp., Ericsson, Hewlett-Packard, ICL, Nixdorf, Olivetti, Philips, Siemens, and Unisys.

PL/I draft available

X3, the Accredited Standards Committee on Information Processing Systems, began a public review and comment period on the revised X3.74-198x, General Purpose Subset of PL/I. The comment period ends April 12.

The revision has been extended to reflect features incorporated in recent implementations and to let the language support a broader range of implementations and applications.

Copies can be obtained for $75 each by calling Global Engineering Documents at (800) 854-7179.

Pascal comment periods

X3, the Accredited Standards Committee on Information Processing Systems, began a public review and comment period on two Pascal draft proposals.

The first proposal is ANSI X3.160-198x, Extended Pascal. The comment period ends May 2.

The standard defines where Pascal may be reasonably extended in a machine-independent and unambiguous manner consistent with existing practice. It will cover Pascal extensions compatible with the Pascal language standard ANSI/IEEE X3.97-1983. The major additions are modularity, separate compilation, dynamic generation of variable sizes and types, improved string and file handling, value initialization, and relaxed declaration order.

The second proposal is ANSI X3.124.2-198x, Pascal Language Binding of the Graphical Kernel System. The comment period ends May 16.

The standard specifies a language-independent nucleus of a graphics system and its language-dependent layer for Pascal.

Copies can be obtained for $35 each by calling Global Engineering Documents at (800) 854-7179.

SQA planning guide


It should be particularly useful to

- (1) a quality assurance person responsible for developing or implementing a software quality assurance plan for a project,

- (2) a software development manager who wants to initiate software quality assurance procedures on a project,

- (3) a purchaser or user of a software product who wants to evaluate a software seller's quality assurance plan or to specify one,

- (4) an independent evaluator, such as an EDP auditor, and

- (5) the person accountable for the implementation of a software quality assurance plan.

The guide can be ordered from the IEEE Service Center, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855.