Writing Quality Requirements[1]

Karl E. Wiegers
Process Impact
www.processimpact.com

It looks like your project is off to a good start. The team got some customers involved in the requirements elicitation stage and you actually wrote a software requirements specification. The spec was kind of big, but the customers signed off on it so it must be okay.

Now you are designing one of the features and you've found some problems with the requirements. You can interpret requirement 15 a couple of different ways. Requirement 9 states precisely the opposite of requirement 21; which should you believe? Requirement 24 is so vague that you haven't got a clue what it means. You just had an hour-long discussion with two other developers about requirement 30 because all three of you thought it meant something different. And the only customer who can clarify these points won't return your calls. You're forced to guess at what many of the requirements mean, and you can expect to do a lot of rework if you guess wrong.

Many software requirements specifications (SRS) are filled with badly written requirements. Because the quality of any product depends on the quality of the raw materials fed into it, poor requirements cannot lead to excellent software. Sadly, few software developers have been educated about how to elicit, analyze, document, and verify the quality of requirements. There aren't many examples of good requirements available to learn from, partly because few projects have good ones to share, and partly because few companies are willing to place their product specifications in the public domain.

This article describes several characteristics of high quality software requirement statements and specifications. We will examine some less-than-perfect requirements from these perspectives and take a stab at rewriting them. I've also included some general tips on how to write good requirements. You might want to evaluate your own project's requirements against these quality criteria. It may be too late to revise them, but you might learn some things that will help your team write better requirements the next time.

Don't expect to create an SRS in which every requirement exhibits all of these desired characteristics. No matter how much you scrub, analyze, review, and refine the requirements, they will never be perfect. However, if you keep these characteristics in mind, you will produce better requirements documents and you will build better products.

Characteristics of Quality Requirement Statements

How can we distinguish good software requirements from those that have problems? This section describes six characteristics individual requirement statements should exhibit, while the next section presents desirable characteristics of the SRS document as a whole. A formal inspection of the SRS by project stakeholders who represent different perspectives is one way to

---

[1] This paper was originally published in *Software Development*, May 1999. It is reprinted (with modifications) with permission from *Software Development* magazine.

determine whether each requirement has these desired attributes. Another powerful quality technique is to write test cases against the requirements before you cut a single line of code. Test cases crystallize your vision of the product's behavior as specified in the requirements and can reveal fuzziness, omissions, and ambiguities.

Correct. Each requirement must accurately describe the functionality to be delivered. The reference for correctness is the source of the requirement, such as an actual customer or a higher-level system requirements specification. A software requirement that conflicts with a corresponding system requirement is not correct (of course, the system specification could itself be incorrect).

Only user representatives can determine the correctness of user requirements, which is why it is essential to include them, or their close surrogates, in inspections of the requirements. Requirements inspections that do not involve users can lead to developers saying, "That doesn't make sense. This is probably what they meant." This is also known as "guessing."

Feasible. It must be possible to implement each requirement within the known capabilities and limitations of the system and its environment. To avoid infeasible requirements, have a developer work with the requirements analysts or marketing personnel throughout the elicitation process. This developer can provide a reality check on what can and cannot be done technically, and what can be done only at excessive cost or with other tradeoffs.

Necessary. Each requirement should document something the customers really need or something that is required for conformance to an external requirement, an external interface, or a standard. Another way to think of "necessary" is that each requirement originated from a source you recognize as having the authority to specify requirements. Trace each requirement back to its origin, such as a use case, system requirement, regulation, or some other voice-of-the-customer input. If you cannot identify the origin, perhaps the requirement is an example of "gold plating" and is not really necessary.

Prioritized. Assign an implementation priority to each requirement, feature, or use case to indicate how essential it is to include it in a particular product release. Customers or their surrogates have the lion's share of the responsibility for establishing priorities. If all the requirements are regarded as equally important, the project manager is less able to react to new requirements added during development, budget cuts, schedule overruns, or the departure of a team member. Priority is a function of the value provided to the customer, the relative cost of implementation, and the relative technical risk associated with implementation.

I use three levels of priority. High priority means the requirement must be incorporated in the next product release. Medium priority means the requirement is necessary but it can be deferred to a later release if necessary. Low priority means it would be nice to have, but we realize it might have to be dropped if we have insufficient time or resources.

Unambiguous. The reader of a requirement statement should be able to draw only one interpretation of it. Also, multiple readers of a requirement should arrive at the same interpretation. Natural language is highly prone to ambiguity. Avoid subjective words like user-friendly, easy, simple, rapid, efficient, several, state-of-the-art, improved, maximize, and minimize. Words that are clear to the SRS author may not be clear to readers. Write each requirement in succinct, simple, straightforward language of the user domain, not in computerese. Effective ways to reveal ambiguity include formal inspections of the requirements specifications, writing test cases from requirements, and creating user scenarios that illustrate the expected behavior of a specific portion of the product.

Verifiable. See whether you can devise tests or use other verification approaches, such as inspection or demonstration, to determine whether each requirement is properly implemented in the product. If a requirement is not verifiable, determining whether it was correctly implemented is a matter of opinion. Requirements that are not consistent, feasible, or unambiguous also are not verifiable. Any requirement that says the product shall "support" something is not verifiable.

Characteristics of Quality Requirements Specifications

A complete SRS is more than a long list of functional requirements. It also includes external interface descriptions and nonfunctional requirements such as quality attributes and performance expectations. Look for the following characteristics of a high quality SRS.

Complete. No requirements or necessary information should be missing. Completeness is also a desired characteristic of an individual requirement. It is hard to spot missing requirements because they aren't there. Organize the requirements hierarchically in the SRS to help reviewers understand the structure of the functionality described, so it will be easier for them to tell if something is missing.

If you focus on user tasks rather than on system functions during requirements elicitation, you are less likely both to overlook requirements and to include requirements that aren't really necessary. The use case method works well for this purpose. Graphical analysis models that represent different views of the requirements can also reveal incompleteness.

If you know you are lacking certain information, use "TBD" ("to be determined") as a standard flag to highlight these gaps. Resolve all TBDs from a given set of the requirements before you proceed with construction of that part of the product.

Consistent. Consistent requirements do not conflict with other software requirements or with higher level (system or business) requirements. Disagreements among requirements must be resolved before development can proceed. You may not know which (if any) is correct until you do some research. Be careful when modifying the requirements, as inconsistencies can slip in undetected if you review only the specific change and not any related requirements.

Modifiable. You must be able to revise the SRS when necessary and maintain a history of changes made to each requirement. This means that each requirement be uniquely labeled and expressed separately from other requirements so you can refer to it unambiguously. You can make an SRS more modifiable by organizing it so that related requirements are grouped together, and by creating a table of contents, index, and cross-reference listing.

Traceable. You should be able to link each software requirement to its source, which could be a higher-level system requirement, a use case, or a voice-of-the-customer statement. Also link each software requirement to the design elements, source code, and test cases that are constructed to implement and verify the requirement. Traceable requirements are uniquely labeled and are written in a structured, fine-grained way, as opposed to large, narrative paragraphs or bullet lists.

Reviewing Requirements for Quality

These descriptions of characteristics of quality requirements are all very fine in the abstract, but what do good requirements really look like? To make these concepts more tangible, let's do a little practice. Following are several requirements adapted from actual projects. Evaluate each one against the preceding quality criteria to see if it has any problems, and rewrite it

in a better way. I'll offer my analysis and improvement suggestions for each one, but you might well come up with a different interpretation. I have an advantage because I know where each requirement came from. Since you and I aren't the real customers, we can only guess at the actual intent of each requirement.

Example #1: *"The product shall provide status messages at regular intervals not less than every 60 seconds."* This requirement is incomplete: what are the status messages and how are they supposed to be displayed to the user? The requirement contains several ambiguities. What part of "the product" are we talking about? Is the interval between status messages really supposed to be at least 60 seconds, so showing a new message every 10 years is okay? Perhaps the intent is to have no more than 60 seconds elapse between messages; would 1 millisecond be too short? The word "every" just confuses the issue. As a result of these problems, the requirement is not verifiable.

Here is one way we could rewrite the requirement to address those shortcomings:

"1. Status Messages.

1.1. The Background Task Manager shall display status messages in a designated area of the user interface at intervals of 60 plus or minus 10 seconds.

1.2. If background task processing is progressing normally, the percentage of the background task processing that has been completed shall be displayed.

1.3. A message shall be displayed when the background task is completed.

1.4. An error message shall be displayed if the background task has stalled."

I split this into multiple requirements because each will require separate test cases and because each should be separately traceable. If several requirements are strung together in a paragraph, it is easy to overlook one during construction or testing.

Example #2: *"The product shall switch between displaying and hiding non-printing characters instantaneously."* Computers cannot do anything instantaneously, so this requirement is not feasible. It is incomplete because it does not state the conditions that trigger the state switch. Is the software making the change on its own under some conditions, or does the user take some action to stimulate the change? Also, what is the scope of the display change within the document: selected text, the entire document, or something else? There is an ambiguity problem, too. Are "non-printing" characters the same as hidden text, or are they attribute tags or control characters of some kind? As a result of these problems this requirement cannot be verified.

This might be a better way to write the requirement: "The user shall be able to toggle between displaying and hiding all HTML markup tags in the document being edited with the activation of a specific triggering condition." Now it is clear that the non-printing characters are HTML markup tags. This requirement does not constrain the design because it does not define the triggering condition. When the designer selects an appropriate triggering condition, you can write specific tests to verify correct operation of this toggle.

Example #3: *"The HTML Parser shall produce an HTML markup error report which allows quick resolution of errors when used by HTML novices."* The word "quick" is ambiguous. The lack of definition of what goes into the error report is a sign of incompleteness. I'm not sure how you would verify this requirement. Find someone who calls herself an HTML novice and see if she can resolve errors quickly enough using the report?

Try this instead: "The HTML Parser shall produce an error report that contains the line number and text of any HTML errors found in the parsed file and a description of each error found. If no errors are found, the error report shall not be produced." Now we know what needs to go into the error report, but we've left it up to the designer to decide what the report should look like. We have also specified an exception condition: if there aren't any errors, don't generate a report.

Example #4: *"Charge numbers should be validated on-line against the master corporate charge number list, if possible."* I give up, what does "if possible" mean? If it's technically feasible? If the master charge number list can be accessed on line? Avoid imprecise words like "should." The customer either needs this functionality or he doesn't. I have seen some requirements specifications in which subtle distinctions are drawn among keywords like "shall", "will", and "should" as a way to indicate priority. I prefer to stick with "shall" as a clear statement of what is intended by the requirement and to explicitly specify the priorities. Here is an improved version of this requirement: "The system shall validate the charge number entered against the on-line master corporate charge number list. If the charge number is not found on the list, an error message shall be displayed and the order shall not be accepted."

The difficulty developers will have in understanding the intent of each poorly written requirement is a strong argument for having both developers and customers review requirements documents before they are approved. Detailed inspection of large requirements documents is not fun. The people I know who have done it, though, feel it was worth every minute they spent on the inspection. It is much cheaper to fix the defects at that stage than later in the development process or when the customer calls.

Guidelines for Writing Quality Requirements

There is no formulaic way to write excellent requirements. It is largely a matter of experience and learning from the requirements problems you have encountered in the past. Here are a few guidelines to keep in mind as you document software requirements.

- Keep sentences and paragraphs short. Use the active voice. Use proper grammar, spelling, and punctuation. Use terms consistently and define them in a glossary or data dictionary.

- To see if a requirement statement is sufficiently well defined, read it from the developer's perspective. Mentally add the phrase, "call me when you're done" to the end of the requirement and see if that makes you nervous. In other words, would you need additional clarification from the SRS author to understand the requirement well enough to design and implement it? If so, that requirement should be elaborated before proceeding with construction.

- Requirement authors often struggle to find the right level of granularity. Avoid long narrative paragraphs that contain multiple requirements. A helpful granularity guideline is to write individually testable requirements. If you can think of a small number of related tests to verify correct implementation of a requirement, it is probably written at the right level of detail. If you envision many different kinds of tests, perhaps several requirements have been lumped together and should be separated.

- Watch out for multiple requirements that have been aggregated into a single statement. Conjunctions like "and" and "or" in a requirement suggest that several requirements have been combined. Never use "and/or" in a requirement statement.

- Write requirements at a consistent level of detail throughout the document. I have seen requirements specifications that varied widely in their scope. For example, “A valid color code shall be R for red” and “A valid color code shall be G for green” might be split out as separate requirements, while “The product shall respond to editing directives entered by voice” describes an entire subsystem, not a single functional requirement.

- Avoid stating requirements redundantly in the SRS. While including the same requirement in multiple places may make the document easier to read, it also makes maintenance of the document more difficult. The multiple instances of the requirement all have to be updated at the same time, lest an inconsistency creep in.

If you observe these guidelines and if you review the requirements formally and informally, early and often, your requirements will provide a better foundation for product construction, system testing, and ultimate customer satisfaction. And remember that without high quality requirements, software is like a box of chocolates: you never know what you’re going to get.