jama software

Writing High-Quality Requirements

By Requirements Expert Karl Wiegers

> **Writing requirements is hard.** There is no simple, formulaic approach to software specification. Requirements management expert Karl Wiegers explains high-quality requirements begin with proper grammar, accurate spelling, well-constructed sentences, and a logical organization.

This white paper, adapted from my book *More About Software Requirements*, presents numerous style guidelines to keep in mind when writing functional requirements. I'm not a fan of arbitrary rules about writing requirements. Some I've heard are:

- A requirement may not contain the word *and.* An *and* indicates the presence of two requirements, which must be separated.
- A requirement may not contain more than one sentence.
- A requirement may not contain more than 22 words.

These sorts of simplistic rules are intended to help analysts write good requirements, but there are too many cases in which they don't constitute good advice. As you develop your requirements specifications, remember your key objective: clear and effective communication among the project stakeholders.

I Shall Call This a Requirement

Shall is the traditional keyword for identifying a functional requirement. Functional requirements describe behaviors the system shall exhibit under certain circumstances or actions the system shall let the user take. Some people object to the use of *shall* because it feels stilted. It's not the way people normally talk, at least not outside English period-piece movies. True—but so what? In fact, this is a plus. Using a distinctive word sharply separates a requirement from other information in a specification document. *Shall* signals the presence of a discrete requirement.

Too many requirements specifications use a random mix of different verbs: *shall, must, should, could, would, is recommended, is desirable, is requested, will, can, may, and migh*t. Many of these words are used interchangeably in casual conversation, but this can become confusing in a written specification. The reader is left to wonder whether there's a subtle but important distinction between these various keywords. Does *must* have some different connotation than *can*? Does *might* (which conveys a sense of possibility in ordinary speech) mean the same thing as *may* (which conveys a sense of permission)? I've also heard of conventions in which *shall* identifies a requirement, but *will* indicates a design statement and *must* signifies a constraint. Oh my.

Some organizations follow a convention I find risky. In this scheme, *shall* indicates a function that is required, *should* means that the function is desired, and may indicates that the function being described is optional. This raises two problems. First, two concepts are being combined: the statement of intended functionality and the relative priority of that functionality. The second problem is that the priority information is being communicated using words that have similar meanings in everyday conversation.

> **Avoid "should", "may," "might," and similar words that don't make it clear whether the statement is a requirement.**

My preference is to use the keyword *shall* to identify functional requirements whenever possible. Avoid *should, may, might,* and similar words that don't make it clear whether the statement is a requirement. My colleague Brian Lawrence suggests that you replace *should* with *probably won't* and see if that would be all right with the customer. It probably won't.

A requirement in the form, "The system should do X" can be restated in the form, "When Y happens, the system shall do X." And instead of using the *shall–should–may* convention to communicate priority, write requirements as follows:

1. "The system shall … [Priority = High]."
2. "The system shall … [Priority = Medium]."
3. "The system shall … [Priority = Low]."

The goal of clear and unambiguous communication is elusive when requirements writers use a mix of nearly synonymous verbs and expect all readers to reach the same conclusions about what they're trying to say. Frankly, I don't understand the objection to *shall*. But if you don't like it, pick an alternative word—such as *must*—and use it consistently.

System Perspective or User Perspective?

Various conventions for writing functional requirements exist. Some people believe that requirements should describe only the system's behavior, because "the system" is what you create by implementing all the functional requirements. However, I think it's appropriate to write functional requirements from either the system's perspective or the user's perspective. Use whichever structure offers the clearest communication in a given situation.

Requirements written from the system's perspective conform to the following general structure:

Conditions: "When [some conditions are true] ..."

Result: "... the system shall [do something]"

Qualifier: "... [response time goal or quality objective]."

The "conditions" part of the requirement could reflect an event that triggers the system to respond in some way. Here's a simple example, from an information system for ordering meals online from a company cafeteria:

> *When the patron indicates that he does not wish to order any more food items, the system shall display all food items ordered, the individual food item prices, and the total payment amount within 1 second.*

This requirement describes an event that the system can detect, followed by the action the system takes in response to that event. This requirement also includes a performance goal, the 1-second response time. This element constitutes a nonfunctional requirement associated with this specific bit of system functionality.

When stating such performance goals, it's important to make clear whether they are critical values or merely desirable targets. Is the system acceptable if it takes 1.2 seconds to display the order details? How about 10 seconds? Precise response times are more critical for real-time hardware systems than for information systems.

In some cases, it makes more sense to describe actions that the system will let the user take under particular circumstances. When writing functional requirements from the user's perspective, keep the following general structure in mind:

User: "The [user class or actor name] ..."

Result: "... shall be able to [do something] ..."

Object: "... [to something].

Qualifier: [response time goal or quality objective]

Whenever possible, refer to the affected user class by name, rather than saying just user. Here's an illustration of a functional requirement written from the user's perspective:

> *The patron shall be able to reorder any meal he had ordered within the previous six months, provided that all food items in that order are available on the menu for the meal date.*

Note that these examples are written in the active voice. They explicitly identify the entity—the system or a specific user type—that takes each action. Most of the functional requirements I read are written in passive voice:

Passive: "When the output state changes, it is logged in the event log."

Whenever you can, recast such requirements in the much clearer active voice:

Active: "When the output state changes, the system shall record the new state and the time of the state change in the event log."

With active voice, the reader doesn't have to deduce which entity is doing what. The more explicit and precise you can make the requirements, the easier it will be for the readers to understand them and use them to guide the project work they do.

Parent and Child Requirements

When writing requirements in a hierarchical fashion, the business analyst (BA) records a parent requirement and one or more child requirements. The parent requirement is satisfied by implementing all of its children. Here's an illustration of a hierarchical requirement with some problems:

3.4 The requester shall enter a charge number for each chemical ordered.

3.4.1 The system shall validate charge numbers against the master corporate charge number list. If the charge number is invalid, the system shall notify the requester and shall not accept the order.

3.4.2 The charge number entered shall apply to an entire order, not to individual line items in the order.

Notice that this parent requirement, 3.4, is written in the form of a functional requirement. It's not entirely clear how many requirements are represented here: two or three? Also notice that there is a conflict between the parent requirement and one of its child requirements, 3.4.2. If each ordered chemical is a line item, exactly how many charge numbers is the requester supposed to enter?

These sorts of problems disappear if the parent requirement is written in the form of a heading or title instead of in the form of a functional requirement. Consider using this style whenever you have a set of child requirements that, in the aggregate, constitute a parent requirement. Following is an improved version of the preceding example:

3.4 Charge Numbers

3.4.1 The requester shall enter a charge number for each chemical in an order.

3.4.2 The system shall validate charge numbers against the master corporate charge number list. If the charge number is not found on this list, the system shall notify the requester and shall not accept the order.

What Was That Again?

Ambiguity is the great bugaboo of software requirements. Ambiguity shows up in two forms. One form I can catch myself. I read a requirement and realize that I can interpret it in more than one way. I don't know which interpretation is correct, but at least I caught the ambiguity.

The other type of ambiguity is much harder to spot. Suppose the BA gives the requirements specification to several reviewers. The reviewers encounter an ambiguous requirement that makes sense to each of them but also means something different to each of them. The reviewers all report back, "These requirements are fine." They didn't find the ambiguity because each reviewer knows only his or her own interpretation of that requirement. Let's look at some sources of ambiguity to watch for and some suggestions about how to write clearer requirements.

Complex Logic

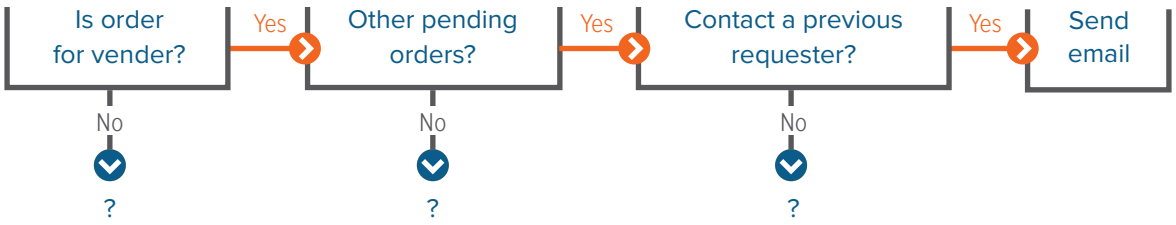
Complex Boolean logic offers many opportunities for ambiguities and missing requirements. Consider the following paragraph:

> *If an order is placed for a chemical to a vendor, the system shall check to see if there are any other pending orders for that chemical. If there are, the system shall display the vendor name, vendor catalog number, and the name of the person who placed each previous order. If the user wishes to contact any person who placed a previous order, the system shall allow the user to send that person an email message.*

This long requirement is difficult to read and contains multiple functionality descriptions that should be split into separate requirements. Plus, it has some gaps. Writing requirements in this style makes it difficult to see whether the outcomes of all the if/then branches are specified. *Else* conditions are often overlooked with this sort of textual representation. Nested *or, and,* and *not* clauses are better represented using a decision table or decision tree.

A decision tree such as that shown in Figure 1 would immediately reveal that the system's behavior is not specified if there are no pending orders for that particular chemical. Other false outcomes from the decisions are also unspecified. Implicitly, perhaps the reader will conclude that the system should do nothing if the various if conditions described here are false, but that's an assumption forced by the incompleteness.

Figure 1. Sample Decision Tree for Complex Logic



Expressions involving compound operators—such as "IF this AND (that OR the other)"—are even more confusing:

> *If the amount of the cash refund is less than $50 or the amount of the cash refund is $50 or greater and the current user is a supervisor, then the system shall open the cash register drawer. Making this requirement understandable and unambiguous requires either parentheses (awkward) or splitting into multiple requirements (better).*

Negative Requirements

Negative (or inverse) requirements are another source of confusion. Try to recast inverse requirements in a positive sense, to state what the system will do under certain circumstances. Table 1 states several functional requirements, all drawn from actual projects, that contain negation, along with possible ways to rewrite them in a positive sense. I've also restated these passive-voice requirements into the less ambiguous active voice, which clearly shows what entity is taking each action. Note that changing a negative requirement into a positive one often requires inserting the word *only* to identify the conditions that permit the system action being described to take place. In the third example, note also the ambiguity between cannot (as in "not able to") and may not (as in "not permitted to").

Table 1: Removing Negation from Functional Requirements

BEFORE	AFTER
All user with three or more accounts should not be migrated.	The system shall migrate only users having fewer than three accounts.
The registration process will default to International English and will not present a localized experience until country and language are selected.	The registration process shall default to International English. After the user selects the country and language, the registration process shall present a localized experience.
A domain name cannot be transferred to another registrar during the registration grace period.	The domain administrator may transfer a domain name to another registrar only after the registration grace period.
The PC administrator will not have the ability to change the FZL-Web user.	Only the system administrator shall be able to change the FZL-Web user.

Avoid double and triple negatives in all circumstances. Consider this example:

> *Training rewards and points will not be visible to users who cannot participate in training rewards.*

We can rephrase this double negative into a positive statement that's easier to understand:

> *The system shall display training rewards and points only to users who are permitted to participate in training rewards.*

The following is another illustration of recasting a double negative into a positive using an only constraint. The original requirement said:

> *Users who are delivered service without being authenticated should not generate accounting records.*

Let's state it as a positive action that the developer can actually implement:

> *The system shall generate accounting records only for users who are delivered service after being authenticated.*

Multiple negations can lead to ambiguous requirements, as illustrated here:

> *Records, however, should not fail validation if these attributes are not present at all.*

Does this mean that the records should fail validation if the attributes are present? Probably not. The context might make the meaning of this requirement clearer, but as it's written, it raises a question in my mind. The main message here is to air on the positive side when writing requirements.

Omissions

When requirements lack important bits of information, it's unlikely that all readers will interpret them in the same way unless they make precisely the same assumptions. For instance, a functional requirement might describe a behavior without identifying the triggering cause that leads to that behavior:

> *The system shall generate an error report and forward it to the user.*

This requirement doesn't identify the stimulus that leads the system to produce the error report. Another common mistake involves missing descriptions of how possible exceptions should be handled. In the previous example, what should happen if no errors occur during the process being described? It's unspecified, thereby leaving it up to the developer to decide what to do. Options include:

- Do nothing (an assumed default perhaps).
- Present a "Congratulations! No errors found." Message, but not generate a report.
- Generate an empty report and forward it to the user.
- Generate a report stating that no errors were found and forward it to the user.

Perhaps we add the following requirement to address the case in which no errors are encountered:

> *If parsing is successful, the system shall not generate an error report.*

This is another description of the system doing nothing. It would be better to state what the system will do if no error is encountered, even if it is to simply continue the processing.

When requirements lack important bits of information, it's unlikely that all readers will interpret them in the same way unless they make precisely the same assumptions.

Another kind of incompleteness occurs when requirements describe system behaviors that involve some type of symmetry. Suppose you're specifying the functional requirements for a bookmark feature in a web browser. You might write:

> *The system shall display the user's defined bookmarks in a collapsible hierarchical tree structure.*

So, the user can collapse the bookmark tree, but what if he wants to expand it again? It's easy to overlook that sort of symmetrical or reverse operation. To remedy this, either you could add a second requirement stating that the tree can be expanded or you could alter this requirement to say "... in a collapsible and expandable hierarchical tree structure."

If you omit the reverse operation, the customer and the BA might assume that the missing portion of the symmetrical requirement is implied. If you request an undo capability, of course you want a redo capability as well, right? But implicit requirements make me nervous. They involve too many assumptions about the knowledge and thought processes that other stakeholders must have to ensure that we all get what we expect in the final product. I know of an organization that developed its own tool for editing and storing source code in a database, with no written requirements. Unfortunately, they forgot to include the ability to print the contents of the database. The team members no doubt assumed that a printing function would be included so didn't even think to mention it. They didn't mention it, and they didn't get it.

Boundaries

Boundary values in numerical ranges provide additional opportunities for creating ambiguity, and they are good places to look for missing requirements. Suppose you're writing software for a point-of-sale system and you need to comply with a business rule that states, "Only supervisors may issue cash refunds greater than $50." An analyst might derive several functional requirements from that business rule, such as the following:

1. If the amount of the cash refund is less than $50, the system shall open the cash register drawer.
2. If the amount of the cash refund is more than $50 and the user is a supervisor, the system shall open the cash register drawer. If the user is not a supervisor, the system shall display a message: "Call a supervisor for this transaction."

But what if the amount of the cash refund is exactly $50? Is this a third, unspecified case? Or is it one of the two cases already described? If so, which one? Such ambiguity forces the developer either to make his best guess or to track down someone who can answer the question definitively. This is an example of the BA generating an inconsistency between a higher-level piece of information—the business rule—and the functional requirements derived from it.

You can resolve boundary ambiguities in one of two ways. The previous requirement #1 could be rewritten as, "If the amount of the cash refund is less than or equal to $50, the system shall open the cash register drawer." This preserves the original intent of the business rule and eliminates the ambiguity. Alternatively, you could use the words inclusive and exclusive to explicitly indicate whether the endpoints of a numerical range are considered to lie within the range or outside the range. To illustrate with a different example, you might say, "The system shall calculate a 20% discount on orders of 6 to 10 units, inclusive." This wording makes it perfectly clear that both endpoints of the range, 6 and 10, lie within the range subject to the 20-percent price discount.

You still need to review a set of similar requirements to make sure the range endpoints don't overlap, though. For example, note the inconsistency between the following two requirements:

1. The system shall calculate a 20% discount on orders of 6 to 10 units, inclusive.
2. The system shall calculate a 30% discount on orders of 10 to 20 units, inclusive.

The boundary value of 10 is incorrectly included in both ranges. Using a table to show this sort of information is concise and makes these kinds of errors more evident:

Units Purchased	Discount Percentage
1-5	0
6-10	20
11-20	30
21+	40

Synonyms

I once reviewed some requirements for software that controlled several analytical chemistry instruments in a laboratory. In some places, the analyst who wrote the specification referred to "chemical samples," and in other places she referred to "runs." I asked her about the difference between a sample and a run. "They're really the same thing," she replied. I suggested she pick one term and stick to it. Whenever I read a document that uses slightly different terms to refer to the same item, I have to check with someone to ascertain whether they are truly synonyms. Place such definitions in a shared glossary so that team members can use them consistently throughout the project and perhaps even across multiple projects.

Elsewhere in that same SRS, the author had used three terms that I thought might be synonyms. When I inquired, I learned that they had subtly different meanings. Define such terms in your project glossary to ensure that all readers can reach the same understanding of the terms.

Pronouns

My mother is a known pronoun abuser. She will say something like, "He said he'd bring that down as soon as he was done with it," and I'll have no idea who or what she is talking about. Pronouns can also be a source of confusion in a requirements specification. Be certain that the antecedent is crystal clear whenever you employ a pronoun. If you use a word such as *this* or *that*, there should be no confusion in the reader's mind about what you're referring to.

The abbreviations i.e. and e.g.

Another ambiguity risk involves using abbreviations that some readers might misconstrue. A common point of confusion is the use of *i.e.* versus *e.g.* Consider the following requirement from an actual specification:

> *The program needs to have a means of allowing the operator to manually activate certain portions of the process in the event a mistake is made (i.e., activate the valve set to apply pressure or vacuum, set pressures, and activate the temperature chamber).*

The abbreviation *i.e.* stands for the Latin phrase *id est*, which means "that is". The abbreviation *e.g.* stands for the Latin phrase *exempli gratia*, which means "for example". These two abbreviations are so commonly confused that I don't trust their use in a requirements specification unless I'm positive that the author understands the difference. In the previous example, the use of i.e. indicates that the following list itemizes all portions of the process that require a means of manual activation. However, if the author really meant for these to be just examples—a portion of that set—he should have used **e.g.** That way, the reader knows that more such manual activations could be needed. Unfortunately, the reader won't have any idea how many more activations might be needed or just what those activations are from this requirement. It's essential to make it clear whether you are presenting a complete list of items or just an illustrative subset. I suggest explicitly saying *for example* instead of *e.g.* so every reader knows what you mean.

A/B

Some specification writers use an A/B writing construct, as in the following example:

> *Prior to operator intervention, a snapshot of this data should be recorded in an audit/history table.*

What exactly does this mean? Is this requirement referring to an audit table, a history table, a history of audits, or an audit of histories? Are both kinds of information stored in the same table, or are audits the same as histories, or what? Other than and/or, read/write, and a few others, the A/B construct is rarely used in formal writing because it is so ambiguous. When I see that construct, I can think of five possible interpretations, but I don't know which one is correct in a given situation:

- A is the same as B. (If A and B are synonyms, use just one term consistently.)
- Both A and B. (Use the explicit conjunction and.)
- A or B. (Use the explicit conjunction or.)
- A is the opposite of B, as in "approving/disapproving changes." (Use the conjunctions and and or as appropriate to convey the correct meaning.)
- "I'm not sure just what I'm thinking here, so I'll leave it up to each reader to decide what he or she thinks this means." (Decide exactly what you intend to say, and choose the right words.)

Similar-Sounding Words

Writers sometimes write one word but mean another. As an illustration, I often hear people say, "I'll flush out that specification some more," when they really mean, "I'll flesh out that specification some more." Hunters flush their prey from their hiding places, but analysts flesh out their requirements to give them more substance. And consider the following example, drawn from an actual SRS for a telephony product:

> *Special Day caller tunes (default) will take priority over all configured individual caller settings that a customer has selected. However, if an individual has been assigned a Special Day caller tune for the same date, this will overwrite the Special Day caller tune.*

You overwrite a piece of data, but you override a default value. In this context, either interpretation is potentially correct, so it's imperative that the author chooses the right word. Watch out for these common types of errors, which sometimes arise from mispronunciations in speech. Keep a dictionary handy so that you can be sure which word to use. A useful reference for common word usage errors in English is provided by Paul Brians at http://www.wsu.edu/~brians/errors/errors.html.

Adverbs

Words that end in -ly often are ambiguous. They might describe some desirable property of the product, but exactly what is desired is left to each reader's interpretation. Here are some real examples of ineffective adverb usage in requirements specifications:

- Provide a reasonably predictable end-user experience.
- Offer significantly better download times.
- Optimize upload and download to perform quickly.
- Performance for these users should broadly match those for ...
- Downloading this file should complete in approximately 15 minutes.
- Exposing information appropriately ...
- Allows the user to edit his interests and possibly search results ...
- Request formats sent by customers must be clearly defined.
- Subscribers who are changing content selection (effectively a subset of the currently subscribed subscribers) ...

- Generally incurs a “per unit” cost …
- To enable remedial action to be initiated in a timely manner …
- … as expediently as possible …
- Occasionally (not very frequently) there will be an error condition …

Some other adverbs to use with caution are *directly, easily, frequently, ideally, instantaneously, normally, optionally, periodically, preferably, rapidly, transparently, typically,* and *usually*. Try to be specific when describing the intended product characteristics so that all readers will share a common vision of what result they will have when they’re done.

In Summary

You won’t learn how to write good requirements from reading a book on software requirements engineering or a book on technical writing. You need practice. Write requirements to the best of your ability, and then enlist some of your colleagues to review them. Constructive feedback from reviewers can help anyone become a better writer. In fact, it’s essential. Requirements quality is in the eye of the reader—not the author—of the requirements. No matter how fine the author thinks the requirements are, the ultimate arbiters are those who must base their own work on those requirements.

ABOUT KARL WIEGERS

Karl has provided training and consulting services worldwide on many aspects of software development, management and process improvement. He has authored five technical books, including *Software Requirements*, and written more than 175 articles. Prior to starting Process Impact in 1997, he spent 18 years at Eastman Kodak Company. His responsibilities there included experience as a photographic research scientist, software applications developer, software manager and software process and quality improvement leader. Karl has led process improvement activities in small application development groups, Kodak’s internet development group and a division of 500 software engineers developing embedded and host-based digital imaging software products.

ABOUT JAMA SOFTWARE

Jama Software brings innovative analytics, solutions and insights to companies creating complex products and mission-critical software systems. With expanded product and service capabilities, the Jama Product Development Platform empowers large enterprises to accelerate development time, mitigate risk, slash complexity and verify regulatory compliance.

Representing the forefront of modern development, its rapidly growing customer base of more than 600 organizations — including SpaceX, NASA, Thales and Caterpillar — use Jama Software to streamline processes and bring complex products to market. Through Predictive Product Development, Jama equips its customers to make the most of their revenue potential and achieve ongoing competitive advantages.