Better Communication Through Better Requirements

Michael J. Hillelsohn
Software Performance Systems

Everyone involved with a software development effort needs to have the same understanding of the meaning of the requirements for the application under development. This article describes several techniques that can be used during analysis to assure that all stakeholders reach a common level of understanding.

What makes a requirement effective? The question hangs in the air of the requirements class I am teaching. This is how I start the class, with this simple question. Participants eventually, cautiously give answers that cover the usual array of what makes for a good requirement: unambiguous, testable, clear statement of need, measurable, functionally worded, etc. All correct, but rarely do I get the answer I am really looking for: An effective requirement communicates clearly to all parties who read/ hear it what a piece of software needs to do to satisfy the user's expectations. When your goal for well-written requirements is to communicate clearly, then you can use some techniques to enhance the communicability of the requirements.

The entire requirements definition process is, intrinsically, an ongoing communications process. The customer states the problem that needs to be solved, then explains the job that system users will perform when solving the problem. The developer translates these statements into functional requirements then asks the customer, "Did I understand you correctly?"

Through a series of successive approximations, the developer's documentation of the required system functionality closely approximates the customer's expectation of operational system performance. Yet in some instances, the communication process breaks down. By applying some simple, very specific techniques, the risk of misinterpretation is effectively reduced along with the resulting expensive rework later during the project's development life cycle. Here are four techniques that Software Performance Systems uses to foster communication during the requirements definition process:

- Train the requirements analysis team on project standards.
- Build a requirements reserved word list.
- Clearly define quality requirements with the customer.
- Conduct requirements *scrubs*.

Used by all parties in the requirements definition process, and regardless of the form of requirements documentation, these techniques enhance communication throughout the project and assure quality output.

Train the Requirements Team on Project Standards

Project standards for requirements include processes and procedures for requirements management activities as well as formats and templates for outputs. Using an event-driven learning (EDL) approach, short training experiences are interjected in the process immediately prior to executing a process step. The training experiences focus on performing the next step in the requirements elicitation, documentation, and management process so that everyone is *on the same page* when the process activity takes place. It is particularly helpful if both the customer and developer participate in these training experiences. That way, the customer has an opportunity to influence the way the activity is carried out in his or her organization.

The process for developing and conducting these training sessions starts with the selection of requirements' formats. Involving the customer in template selection communicates what to expect as a final product of the requirements definition effort. As usual with process definition, looking at the output of the requirements definition process provides an indication of what the templates need to contain. For example, if the output is going into a requirements management tool then a decision needs to be made whether to let the tool number the requirements or whether to assign numbers to each requirement regardless of the tool's numbering strategy.

In general, the templates will come either from the artifact templates of the selected software development life cycle or an industry standard such as IEEE 830-1998. In either case, there is a likelihood the template will be tailored to suit the project needs such as defining additional attributes (i.e., priority, source, status, planned release, etc.) for the requirements. Once the templates are defined, then EDL is used to communicate the formats and contents to all participants. With all stakeholders on the project working from a common set of templates, everyone knows what to expect, in terms of artifacts, and is familiar with the contents of each section.

A variety of training experiences need to be tailored to the expected artifacts from the requirements management process. If a requirements management plan is being prepared, the training may consist of little more than reviewing the template and discussing sections of the document with the authors before it is written. If the interim artifact is going to be a use-case or software requirements specification (SRS), then more extensive training experiences are warranted.

The classes will have much similar content such as properly formatting the requirements statement as a single action without using any conjunctions, except in the case of Boolean logic. Instruction is tailored to conventions used for each of the different artifacts. For example, to easily interpret the flows in a use case the labeling conventions are defined to uniquely identify each action in the use case. Thus the use-case writer encloses alternate flow identifiers in parentheses next to the primary flow statement. Figure 1 shows some of the conventions adopted for writing use cases, and taught in the Defining Requirements with Use Cases class.

Adopting such conventions — and ensuring that everyone on the project knows how they are used and what they mean — allows the project to develop its own notation while ensuring that everyone can read and unambiguously understand the artifacts. During training, the requirements analysts have an opportunity to apply these conventions as they practice writing use cases immediately before they write the actual ones. SRS authors, on the other hand, spend more of their training time practicing writing and decomposing properly prepared, testable *shall* statements.

Regardless of the class, the emphasis is on writing requirements that clearly convey to users, testers, and designers what is to happen on the system. The total quality management concept of preparing output that is usable by internal and external customers is especially valid when training people to write better requirements. All potential requirements' *customers* are considered as the requirements are written and reviewed.

Build a Requirements Reserved Word List

Words and terminology are the primary communication among people. Word-based communication can be imprecise because the same word heard/read by different people can have different meanings, different words can mean the same activity or thing, and different groups of people have their own jargon. Unified Modeling Language attempts to address these issues with heavy use of diagrams with well-defined components; however, this approach does not help with textually expressed requirements.

The inclusion of a glossary as an artifact in the Rational Unified Process defines domain-specific terminology, but does not provide for unambiguous expression of what the system shall do when it is built. Some early programming languages either specified or allowed the programmer to define a set of reserved words for the program. They allowed the programmer to communicate unambiguously with the computer to perform specific operations consistently. This concept transfers effectively to requirements engineering.

A reserved word list for requirements contains both general and domain-specific terms. General terms select a single word to describe a specific operation. For example, you can specify that the system needs to append information to a database by saying that it shall *save*, *record*, *capture*, or *store* a data element. A team of requirements analysts may use any or all of these words in their requirements; when the tester, designer, or customer reads the requirements, they may or may not interpret each word differently with different tests, designs, or acceptance criteria for each requirement. It would be clearer to define *record* as the only verb to be used to update a database that maintains historical information for the system. A clearly bounded definition like this also makes the other verbs available for things like temporary and local information storage by the user.

There are no synonyms in the reserved word list because they compromise precision of expression. In case-management operations, for example, the terms *case*, *file*, and *work-item* are often used interchangeably. Sometimes in the user community, these terms are used differently in different parts of the organization. It is critical that a single term is used to define the object that is being manipulated by the system so that everyone knows what the requirement is acting upon. In this instance, discussions with the user community will arrive on a consensus term and its definition. Occasionally the discussions with the user may result in nuanced definitions where multiple terms are used but each has a unique attribute. For example, *case* may be used to refer to the offline version while *file* refers to the online version of the same information.

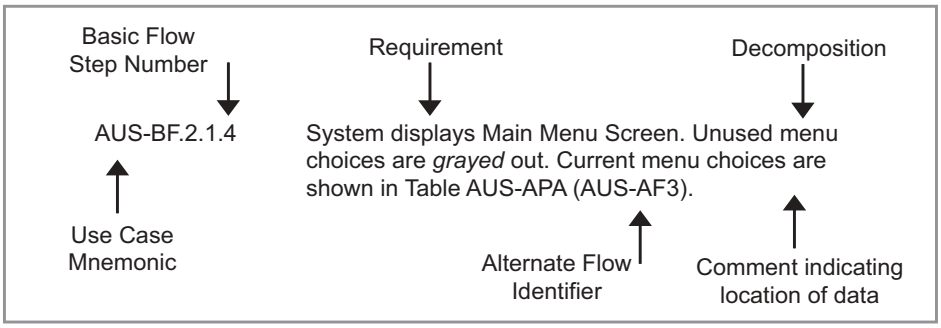

Figure 1: *Defining Standard Notations for Use Cases*

Most words in the reserved word list are verbs with very specific meanings. Table 1 shows three specific terms for user/system interactions (from the user/actor perspective) that clearly communicate the nature of the interaction. The fourth allows for the interaction to be more clearly defined during design. Thus when the interaction is known during requirements definition, it is documented within the requirement/use-case so that it can be confirmed during requirements review cycles. If the interaction is not predetermined, a method of documenting the requirement is available that does not preclude design options.

Adverbs such as *automatically* and *may* are also useful additions to the list; they can be shorthand for describing some common constructs in applications systems. For example, *automatically* can be used before a system action when there is no intervening or triggering action by the user; *may* can indicate that the user action described in the requirement is optional. Constructs such as optional user actions and automatic system actions are easily captured and communicated by designating specific reserved words to represent the ideas. If reserved words are not designated or defined, then the representation of the constructs can become convoluted in requirements documentation.

Occasionally, an analyst may use a word that is not on the reserved word list as a system or user/actor action. This is usually detected during the quality assurance (QA) review of the requirements artifact. In that instance, the reviewer should give the analyst the option of either using a word from the reserved word list, or if there is no term that adequately and accurately describes what is going on, then the analyst is asked to add the term and its definition to the list. Over time, the list will grow to meet the exact needs of the application domain. In general, the reserved word list does not grow too large because systems can only do a limited number of things (e.g., print, display, record, calculate, verify, etc.). The reserved word list is included as an appendix with the documented requirements in the SRS or with the supplementary requirements.

Define Quality Requirements With the Customer

The statement "I know quality when I see it" is not a viable means of defining the quality of software applications. Quality requirements must be defined in the beginning of a project so that they are considerations in the design of the architecture and application features. Software quality factors were defined by the Department of Defense to be used in the acquisition process.

Software Performance Systems uses quality factors as a means for customers and

Table 1: *Using Reserved Words*

Reserved Word	Definition
Enter	A situation where there is a non-specified form for the user to provide information to the system. *Enter* is used where the specific entry method (typing, indicating) is either to be defined during design or when there is more than one way to provide the information. In the latter case, the comment field should contain the acceptable alternate forms by which the user can provide the information.
Indicate	The user makes a binary choice by setting an online Yes/No, On/Off, or other type of flag (e.g., by marking or inserting a symbol such as *comma* [,] in a check box), or otherwise select a setting from an online indicator. A single mouse click or screen touch is a typical method of interaction.
Input	Keyboard entry of information by a user (e.g., typing) requiring an end-of-input signal (i.e., [Enter]).
Select	Choose from a set of options displayed on a list.

Quality Factor	Definition	Components
Efficiency	Relative extent to which computer resources are utilized.	Communication, Processing, Storage
Integrity	Extent to which security protocols are implemented to guard against unauthorized use and access to software and data.	Access Control, Virus Prevention
Reliability	Extent to which the software can perform without any failures.	Accuracy, Anomaly Management, Simplicity
Usability	Relative effort required by the user community to use the application.	Operability, Training, User Support
Correctness	Extent to which the software conforms to its standards and specifications.	Completeness, Traceability, Accuracy, Consistency
Maintainability	Ease of effort associated with finding and fixing a software failure.	Consistency, Visibility, Modularity, Simplicity, Documentation
Verifiability	Relative effort required to test the software operation and performance.	Visibility, Traceability, Documentation, Simplicity
Expandability	Relative effort to increase the application's capabilities and functionality.	Functional Scope, Virtuality, Modularity, Documentation
Interoperability	Ability of the software to function on a variety of platforms and operating systems.	Commonality, Common Functions, Independence, System Compatibility
Portability	Relative effort required to transport the system to another environment.	Independence, Modularity, Documentation
Reusability	Relative ease of using software components, unchanged in other applications.	Application Independence, Clarity, Document Accessibility, Modularity

Table 2: *Selecting Quality Requirements During Analysis*

end users to express how a system satisfies their highest priority, non-functional needs when it is in production. The basis for defining what quality really means to the customer is to have the customer select the quality factors that are most critical to the effort, and then define requirements that set clear, meaningful, attainable, and measurable quality attributes for the application. When these requirements are met, the customer will have a quality product as defined in their own terms for this specific application.

The 11 quality factors shown in Table 2 are derived from the work sponsored by Rome Air Development Center (performed by Boeing Aerospace) in the mid-80s. The user is asked to prioritize the factors since the application cannot consider all of them as the most important. Prioritization involves discussing each of the factors among the group, then each group member selects their three most important factors.

Customers may balk at having to select just a few factors, but the process of prioritization makes them focus on what is critical for the system's success in their environment. For example, although *correctness* is always important, it may not be as critical for a case-management system as it is for a financial system. Likewise *usability* is a higher priority when there will be a large user base than when there are only a few trained users.

After the votes are tallied and presented to the group, another discussion ensues to arrive at a consensus of the three most important quality factors. These discussions may be the first time that members of the customer/user community really try to articulate how quality is defined for their application. While the development team certainly does not ignore factors that are not the highest priority, their importance is secondary; they are not decomposed and measured the same way as the high priority quality factors. By going through the selection and prioritization process, the user community communicates among themselves and to the developer those factors that are most critical and a little less important in the final product.

Quality requirements now need to be defined so everyone agrees on observable criteria for determining if the high-priority quality factors have been achieved. This step requires the user to translate the abstract definition of the quality factor into operationally meaningful terms. When defining reliability, the facilitator (F)/user (U) interaction typically goes something like this:

F: You decided that you need a reliable system. How reliable does it have to be?
U: Oh, very reliable.
F: Can it go down for an hour once a year?
U: Sure, no problem.
F: How about if it goes down once a month? Does that endanger you accomplishing your mission?
U: I suppose once a month may be acceptable, but it depends when.
F: What activity is so important that it absolutely cannot go down during the activity?
U: When agents are uploading case files.
F: Okay, so with the exception of the time when the agents are uploading case files, would it be acceptable if it goes down once a week?
U: That would be the absolute most that we could tolerate, but only if it was down for a very short period of time. And so on.

Now requirements define when the system cannot have a failure and that the mean time between failures is at least 40 operational hours. Continued discussion defines the acceptable time limits on repair/down time under various circumstances and other aspects of system reliability. Shown below are samples of requirements that a specific user group arrived at when defining reliability for their system.

- **QFR.1:** Each iteration of the system shall be verified before it is put into production.
- **QFR.1.1** Each iteration of the system shall have no critical system failures after it is placed in production.
- **QFR.1.2** Each system iteration shall have one or less major defects arise during the next six months in production.

All of these requirements are observable and measurable. Some can be verified during testing, but most must wait until the system is in production to verify that they have been satisfied. They clearly communicate to all parties what the user means by saying the quality system is reliable. The developer uses these requirements during design to specify a system that is most likely to meet these quality requirements (e.g., degree of component architecture, redundancy, etc.). If these and other quality requirements are not communicated to the development team early in the development life cycle, then the probability that the delivered system meets the users' quality expectations is reduced.

The biggest benefit of defining quality requirements in such specific terms during analysis is to communicate these requirements within the user community. It is common that the developer's client and the end-user of the system are different parts of the sponsoring organization. It is critical for system success that the entire sponsoring organization agrees on what constitutes a quality delivered system. Reducing ambiguity in defining quality among all system stakeholders leads to better assurance of achieving customer satisfaction when the system goes into production.

Conduct Requirements Scrubs

Inspections and walkthroughs are proven techniques for early defect detection and are considered to have a significant positive impact on the *cost of quality* of software products. We should do formal inspections on requirements artifacts, but often the potential participants are put off by the formality

of the process. So Software Performance Systems introduced the term scrubbing the requirements, which is less formal but achieves the same end as formal inspections.

During a thorough requirements reading in a group setting, many defects are detected and corrected. In addition, Software Performance Systems often has a navigational prototype available to the participants to help them visualize how the words translate into an actual application.

Where formal inspections rely on a small focused cadre of reviewers (generally set at three to five people), it is imperative that multiple perspectives are represented at requirements scrubs. People present at the scrub should include the following:

- **Producer.** The person(s) who actually wrote the requirements. During the scrub, the producer will read each requirement aloud.
- **Customer.** The requirements are a description of what the customer will receive when the application is delivered, so the customer needs to be present to hear and provide input to the detailed application description. The customer is also the referee when requirements changes are suggested (generally by the user) that may be out of scope.
- **End-User.** The specifics of what can be done with the application are represented by the functionality described in the requirements, so the scrub gives users the opportunity to correct misconceptions about how they do their job. Just a few knowledgeable user representatives who can make decisions for the functional area being reviewed will be productive. Too many users at a scrub leads to long discussions without resolution.
- **Tester.** The tester will determine whether there is enough information present in the requirements to develop test cases to verify that the requirement is adequately satisfied when the application is delivered. It is important that the tester ask for clarification in the presence of the customer and user so that a common understanding about the acceptance criteria for each requirement is established.
- **Quality Assurance.** The scrub is an early opportunity for QA to correct defects related to noncompliance with standards for writing well-crafted requirements. QA can also facilitate the scrub.
- **Technical Lead.** As the user discusses how the requirements are satisfied on the job and the analyst describes the requirements, it is helpful to include technical representation at the scrub to get a feel for how the requirements translate into the real world of the user, and to determine the feasibility of implementing the requirements as stated.
- **Project Manager**. Since the user and/or customer may raise concerns about functionality that may either be in or out of scope for the effort, the project manager may want to attend the scrub. Alternatively, any questions regarding scope may be deferred until consultation with the project manager is possible. In most cases this is preferable rather than discussing scope issues during the scrub.

The scrub is conducted very similarly to a formal inspection. The requirements are distributed to the participants in advance, and they are encouraged to review them and be prepared with comments. In reality, tasking customers and users – key participants in the scrub – to do the review prior to the scrub is problematic. Preparation will make the scrub more efficient, but unlike an inspection it is not cancelled if the participants do not prepare adequately.

At the scrub, each requirement or step in a use case is read aloud and comments are provided immediately after the reading. For a use case, alternative flows are read at the point where they would diverge from the main flow. The comments are recorded either by the producer and QA or a scribe. At this point, the facilitator's task is to keep the discussion focused and brief. It is important to maintain a good tempo at the scrub and not allow it to get bogged down in long discussions on a single requirement.

When the navigational prototype is used at the scrub, the facilitator should steer the discussion and comments away from design issues (how the screen looks) and maintain the focus on requirements and functionality. In general, each session should last for no more than two hours; otherwise, it is hard to stay focused. If more time is needed, break the scrub into multiple sessions. The ideal outcome of each discussion is a reworded requirement that everyone agrees is accurate, verifiable, and feasible. If that end-state cannot be reached quickly during the scrub, defer the discussion for offline resolution.

The dynamics of the scrub foster communication among all the stakeholders in the system. As participants attend multiple scrubs, they get a good appreciation for the overall functionality of the planned application and how the pieces fit together. Like an inspection, scrubs serve an important role in fostering a common understanding and knowledge base about the application early in the product's life cycle. This way misunderstandings can be clarified and issues related to functionality shared among stakeholders, and resolved.

Scrubs also are excellent training vehicles for improving the requirements analyst's knowledge and skills relative to writing well-constructed requirements. At one recent scrub, right after the analyst read the requirement aloud, before anyone could comment, she covered her face with her hands and said, "That was a terrible requirement, I'll fix it." Everyone laughed and the scrub continued. The most recent SRS submitted by this author-analyst, to QA, had only .02 defects per page compared with the .80 defects per page before Software Performance Systems implemented these techniques in 1999.

Conclusion

Generally, the systems built by Software Performance Systems are used by people when they perform their jobs or by the public when they provide input to an agency. If the delivered system is not correct, then the agency is crippled in achieving its mission. That is why Software Performance Systems has focused on the importance of using requirements to assure that communications between the developer and the customer are an accurate reflection of functionality that is required when a system goes into production. Using multiple techniques to enhance the communications process during the inception of a software development effort ensures that the delivered application meets the user's quality expectations and is fit for use by the intended audience.◆

About the Author

Michael J. Hillelsohn is a director of Product Assurance at Software Performance Systems in Arlington, Va. He is a certified quality professional with more than 30 years of experience doing development, management, and performance improvement in software and systems development environments. His multi-disciplinary approach combines quality systems and training expertise to improve the performance of organizations and individuals. Hillelsohn's process-oriented performance-engineering methods facilitate adoption of external frameworks (Capability Maturity Model®, ISO, Baldrige) to improve the quality of organizational products and services.

3141 Fairview Park DR
STE 850
Falls Church,VA 22042
Phone: (703) 839-4055
E-mail: mhillelsohn@goSPS.com
hillelsohn@erols.com