Karl Wiegers Describes 10 Requirements Traps to Avoid[1]

Karl E. Wiegers
Process Impact
www.processimpact.com

The path to quality software begins with excellent requirements. Slighting the processes of requirements development and management is a common cause of software project frustration and failure. This article describes ten common traps that software projects can encounter if team members and customers don't take requirements seriously. I describe several symptoms that might indicate when you're falling victim to each trap, and I offer several solutions to control the problem.

Be aware, though, that none of these solutions will work if you're dealing with unreasonable people who are convinced that writing requirements is time-wasting bureaucratic overhead. To persuade such skeptics, present data such as that from the Standish Group's CHAOS report (http://www.scs.carleton.ca/~beau/PM/Standish-Report.html)—a study of 8,380 IT projects which found that more than half were "challenged," with reduced functionality being delivered over-budget and beyond the estimated schedule. The top three contributing factors on challenged projects were lack of user input (12.8% of projects), incomplete requirements and specifications (12.3%), and changing requirements and specifications (11.8%).

Trap #1: Confusion Over "Requirements"

Symptoms: Even the simple word "requirements" means different things to different people. An executive's notion of "requirements" might be a high-level product concept or business vision, while a developer's "requirements" might look suspiciously like detailed user interface designs. Customer-provided requirements often are really solution ideas. One symptom of potential problems is that project stakeholders refer to "the requirements" with no qualifying adjectives. The project participants therefore will likely have different expectations of how much detail to expect in the requirements.

Another symptom is that the users provide "the requirements," but developers still aren't sure what they're supposed to build. If requirements discussions focus exclusively on functionality, the participants might not understand the various kinds of information that fall under the broad rubric of "requirements." As a consequence, important stakeholder expectations might go unstated and unfulfilled.

Solutions: The first step is to recognize that there are several types of requirements, all legitimate and all necessary. A second step is to educate all project participants about key requirements engineering concepts, terminology, and practices.

I think in terms of three levels of requirements, all of which must be addressed during requirements development (Figure 1). At the top are the *business requirements*, representing the high-level objectives of the organization or customer requesting the system or product. They

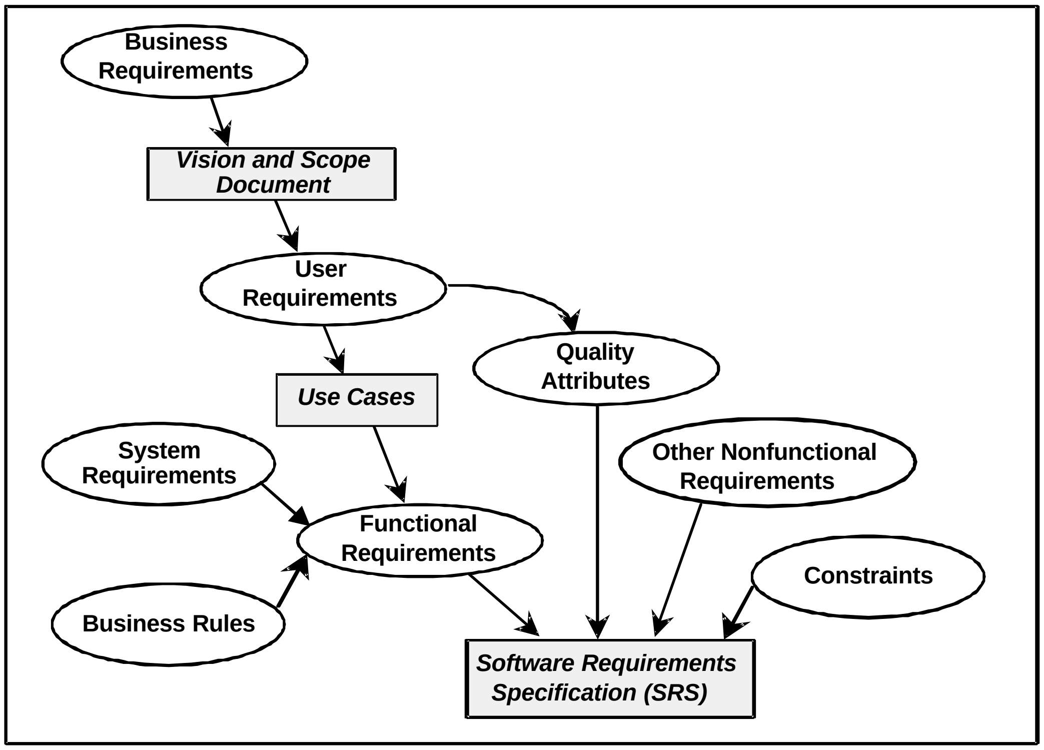
[1] This paper was originally published in *Software Testing & Quality Engineering* magazine (http://www.stqemagazine.com), January/February 2000. It is reprinted (with modifications) with permission from Software Quality Engineering.

describe how the world will be better if the new product is in it. You can record business requirements in a product vision and scope document.

The second level addresses the *user requirements*, which describe the tasks that users must be able to perform using the new product. These are best captured in the form of use cases, which are stories or scenarios of typical interactions between the user and the system. However, the use cases alone often don't provide enough detail for developers to know just what to build. Therefore, you should derive specific software *functional requirements*—the third requirements level—from the use cases. The functional requirements itemize the specific behaviors the software must exhibit.

The software requirements specification (SRS) serves as a container for both the functional requirements and the nonfunctional requirements. The latter include quality attribute goals, performance objectives, business rules, design and implementation constraints, and external interface requirements. Quality attributes (such as usability, efficiency, portability, and maintainability) need to be elicited from users, along with the use cases. (Suggested templates for the SRS, the vision and scope document, and use cases are available at http://www.processimpact.com/goodies.shtml.)

Figure 1. Three Levels of Software Requirements



Trap #2: Inadequate Customer Involvement

Symptoms: Despite considerable evidence that it doesn't work, many projects seem to rely on telepathy as the mechanism for communicating requirements from users to developers. Users sometimes believe that the developers should already know what users need, or that technical stuff like requirements development doesn't apply to users. Often, users claim to be too busy to spend the time it takes to iteratively gather and refine the requirements. (Isn't it funny how we never have time to do things right, but somehow we always find the time to do them over?)

One indication of inadequate customer involvement is that user surrogates (such as user managers, marketing staff, or software developers) supply all of the input to requirements. Another clue is that developers have to make many requirements decisions without adequate information and perspective. If you've overlooked or neglected to gather input from some of the product's likely user classes, someone will be unhappy with the delivered product. On one project I know of, the customers rejected the product as unacceptable the first time they saw it, which was at its initial rollout. This is a strong—but late and painful—indication of inadequate customer involvement in requirements development.

Solutions: Begin by identifying your various user classes. User classes are groups of users who differ in their frequency of using the product, the features they use, their access privilege level, or in other ways. (See "User-Driven Design" by Donald Gause and Brian Lawrence in *Software Testing and Quality Engineering*, January/February 1999, for an excellent discussion of user classes.)

An effective technique is to identify individual "product champions" to represent specific user classes. Product champions collect input from other members of their user class, supply the user requirements, and provide input on quality attributes and requirement priorities.

This approach is particularly valuable when developing systems for internal corporate use; for commercial product development it might be easier to convene focus groups of representative users. Focus group participants can provide a broad range of input on desired product features and characteristics. The individuals you select as user representatives can also evaluate any prototypes you create, and review the SRS for completeness and accuracy. Strive to build a collaborative relationship between your customer representatives and the development team.

Trap #3: Vague and Ambiguous Requirements

Symptoms: Ambiguity is the great bugaboo of software requirements. You've encountered ambiguity if a requirement statement can have several different meanings and you're not sure which is correct. A more insidious form of ambiguity results when multiple readers interpret a requirement in different ways. Each reader concludes that his or her interpretation is correct, and the ambiguity remains undetected until later—when it's more expensive to resolve.

Another hint that your requirements are vague or incomplete is that the SRS is missing information the developers need. If you can't think of test cases to verify whether each requirement was properly implemented, your requirements are not sufficiently well defined. Developers might *assume* that whatever they've been given in the form of requirements is a definitive and complete product description, but this is a risky assumption.

The ultimate symptom of vague requirements is that developers have to ask the analyst or customers many questions, or they have to guess about what is really intended. The extent of this guessing game might not be recognized until the project is far along and implementation has diverged from what is really required. At this point, expensive rework may be needed to bring things back into alignment.

Solutions: Avoid using intrinsically subjective and ambiguous words when you write requirements. Terms like minimize, maximize, optimize, rapid, user-friendly, easy, simple, often, normal, usual, large, intuitive, robust, state-of-the-art, improved, efficient, and flexible are particularly dangerous. Avoid "and/or" and "etc." like the plague. Requirements that include the word "support" are not verifiable; define just what the software must do to "support" something.

It's fine to include "TBD" (to be determined) markers in your SRS to indicate current uncertainties, but make sure you resolve them before proceeding with design and construction.

To ferret out ambiguity, have a team that represents diverse perspectives formally inspect the requirements documents. Suitable inspectors include:

- the analyst who wrote the requirements
- the customer or marketing representative who supplied them (particularly for use case reviews)
- a developer who must implement them
- a tester who must verify them

Begin writing test cases early in requirements development. Writing conceptual test cases against the use cases and functional requirements crystallizes your vision of how the software should behave under certain conditions (see Ross Collard's article "Test Design," *Software Testing and Quality Engineering*, July/August 1999). This practice helps reveal ambiguities and missing information, and it also leads to a requirements document that supports comprehensive test case generation.

Consider developing prototypes; they make the requirements more tangible than does a lifeless textual SRS. Create a partial, preliminary, or possible implementation of a poorly understood portion of the requirements to clarify gaps in your knowledge. Analysis models such as data flow diagrams, entity-relationship diagrams, class and collaboration diagrams, state-transition diagrams, and dialog maps provide alternative and complementary views of requirements that also reveal knowledge gaps.

Trap #4: Unprioritized Requirements

Symptoms: "We don't need to prioritize requirements," said the user representative. "They're all important, or I wouldn't have given them to you." Declaring all requirements to be equally critical deprives the project manager of a way to respond to new requirements and to changes in project realities (staff, schedule, quality goals). If it's not clear which features you could defer during the all-too-common "rapid descoping phase" late in a project, you're at risk from unprioritized requirements.

Another symptom of this trap is that more than 90% of your requirements are classified as high priority. Various stakeholders might interpret "high" priority differently, leading to mismatched expectations about what functionality will be included in the next release. Sometimes developers balk at prioritizing requirements because they don't want to admit they can't do it all in the time available. Often users are also reluctant to prioritize because they fear the developers will automatically restrict the project to the highest priority items and the others will never be implemented. They might be right about that, but the alternatives can include software that is never delivered and having ill-informed people make the priority trade-off decisions.

Solutions: The relative implementation priority is an important attribute of each use case, feature, or individual functional requirement. Align use cases with business requirements, so you know which functionality most strongly supports your key business objectives. Your high-priority use cases might be based on:

- The anticipated frequency or volume of usage
- Satisfying your most favored user classes
- Implementing core business processes
- Functionality demanded for regulatory compliance

If you derived functional requirements from the use case descriptions, this alignment helps you implement the truly essential functionality first. Allocate each requirement or feature to a specific build or release.

Many organizations use a three-level prioritization scale. If you do, define the priority categories clearly to promote consistent classification and common expectations. A more robust solution is to analytically prioritize discretionary requirements, based on their projected customer value and the estimated cost and technical risk associated with construction. (A spreadsheet to assist with this approach is available from http://www.processimpact.com/goodies.shtml.)

Trap #5: Building Functionality No One Uses

Symptoms: I've experienced the frustration of implementing features that users swore they needed, then not seeing anyone use them. I could have spent that development time much more constructively. Beware of customers who don't distinguish glitzy user interface "chrome" from the essential "steel" that must be present for the software to be useful. Also beware of developer gold plating, which adds unnecessary functionality that "the users are just going to love." In short, watch out for proposed functionality that isn't clearly related to known user tasks or to achieving your business goals.

Solutions: Make sure you can trace every functional requirement back to its origin, such as a specific use case, higher-level system requirement, business rule, industry standard, or government regulation. If you don't know where a requirement came from, question whether you really need it. Identify the user classes that will benefit from each feature or use case.

Deriving the functional requirements from use cases is an excellent way to avoid orphan functionality that just seems like a cool idea. Analytically prioritizing the requirements, use cases, or features also helps you avoid this trap. Have customers rate the value of each proposed feature, based on the relative customer benefit provided if it is present—and the relative penalty if it is not. Then have developers estimate the relative cost and risk for each feature. Use the spreadsheet mentioned under Trap #4 to calculate a range of priorities, and avoid those requirements that incur a high cost but provide relatively low value.

Trap #6: Analysis Paralysis

Symptoms: If requirements development seems to go on forever, you might be a victim of analysis paralysis. Though less common than skimping on the requirements process, analysis paralysis results when the viewpoint prevails that construction cannot begin until the SRS is complete and perfect. New versions of the SRS are released so frequently that version numbers resemble IP addresses, and a requirements baseline is never established. All requirements are modeled six ways from Sunday, the entire system is prototyped, and development is held up until all requirement changes cease.

Solutions: Your goal is not to create a perfect SRS, but to develop a set of clearly expressed requirements that permit development to proceed at acceptable risk. If some requirements are uncertain, select an appropriate development lifecycle that will let you implement portions of the requirements as they become well understood. (Some lifecycle choices include the spiral model, staged release, evolutionary prototyping, and time-boxing.) Flag any knowledge gaps in your SRS with "TBD" markers, to indicate that proceeding with construction of those parts of the system is a high-risk activity.

Identify your key decision-makers early in the project, so you know who can resolve issues to let you break out of the paralysis and move ahead with development. Those who must use the requirements for subsequent work (design, coding, testing, writing user documentation) should review them to judge when it's appropriate to proceed with implementation. Model and prototype just the complex or poorly understood parts of the system, not the whole thing. Don't make prototypes more elaborate than necessary to resolve the uncertainties and clarify user needs.

Trap #7: Scope Creep

Symptoms: Most projects face the threat of scope creep, in which new requirements are continually added during development. The Marketing department demands new features that your competitors just released in *their* products. Users keep thinking of more functions to include, additional business processes to support, and critical information they overlooked initially. Typically, project deadlines don't change, no more resources are provided, and nothing is deleted to accommodate the new functionality.

Scope creep is most likely when the product scope was never clearly defined in the first place. If new requirements are proposed, rejected, and resurface later—with ongoing debates about whether they belong in the system—your scope definition is probably inadequate.

Requirement changes that sneak in through the back door, rather than through an established and enforced change control process, lead to the schedule overruns characteristic of scope creep. If Management's sign-off on the requirements documents is just a game or a meaningless ritual, you can expect a continuous wave of changes to batter your project.

Solutions: All projects should expect some requirements growth, and your plans should include buffers to accommodate such natural evolution. The first question you should ask when a new feature, use case, or functional requirement is proposed is: "Is this in scope?" To help you answer this question, document the product's vision and scope and use it as the reference for deciding which proposed functionality to include.

Apparent scope creep often indicates that requirements were missed during elicitation, or that some user classes were overlooked. Using effective requirements gathering methods early on will help you control scope creep. Also, establish a meaningful process for baselining your requirements specifications. All participants must agree on what they are saying when they approve the requirements, and they must understand the costs of making changes in the future. Follow your change control process for *all* changes, recognizing that you might have to renegotiate commitments when you accept new requirements.

Trap #8: Inadequate Change Process

Symptoms: The most glaring symptom of this trap is that your project doesn't have a defined process for dealing with requirements changes. Consequently, new functionality might become evident only during system or beta testing. Even if you have a change process in place, some people might bypass it by talking to their buddies on the development team to get changes incorporated. Developers might implement changes that were already rejected or work on proposed changes before they're approved. Other clues that your change process is deficient are that it's not clear who makes decisions about proposed changes, change decisions aren't communicated to all those affected, and the status of each change request isn't known at all times.

Solutions: Define a practical change control process for your project. You can download a sample change control process from http://www.processimpact.com/goodies.shtml. You can supplement the process with a problem- or issue-tracking tool to collect, track, and communicate changes. However, remember that a *tool* is not a substitute for a *process*. Set up a change control board (CCB) to consider proposed changes at regular intervals and make binding decisions to accept or reject them. (See “How to Control Software Changes” by Ronald Starbuck in *Software Testing and Quality Engineering*, November/December 1999, for more about the CCB.)The CCB shouldn’t be any larger or more formal than necessary to ensure that changes are processed effectively and efficiently. Establish and enforce realistic change control policies. Compare the priority of each proposed requirement change against the body of requirements remaining to be implemented.

Trap #9: Insufficient Change Impact Analysis

Symptoms: Sometimes developers or project managers agree to make suggested changes without carefully thinking through the implications. The change might turn out to be more complex than anticipated, take longer than promised, be technically or economically infeasible, or conflict with other requirements. Such hasty decisions reflect insufficient analysis of the impact of accepting a proposed change. Another indication of inadequate impact analysis is that developers keep finding more affected system components as they implement the change.

Solutions: Before saying “sure, no problem,” systematically analyze the impact of each proposed change. Understand the implications of accepting the change, identify all associated tasks, and estimate the effort and schedule impact. Every change will consume resources, even if it’s not on the project’s critical path. Use requirements traceability information to help you identify all affected system components. Provide estimates of the costs and benefits of each change proposal to the CCB before they make commitments.

Trap #10: Inadequate Version Control

Symptoms: If accepted changes aren’t incorporated into the SRS periodically, project participants won’t be sure what all is in the requirements baseline at any time. If team members can’t distinguish different versions of the requirements documents with confidence, your version control practices are falling short. A developer might implement a canceled feature because she didn’t receive an updated SRS. I know of a project that experienced a spate of spurious defect reports because the system testers were testing against an obsolete version of the SRS.

Using the document’s date to distinguish versions is risky. The dates might be the same but the documents may be different (if you made changes more than once in a day), and identical documents can have different “date printed” labels. If you don’t have a reliable change history for your SRS, and earlier document versions are gone forever, you’re caught in this trap.

Solutions: Periodically merge approved changes into the SRS and communicate the revised SRS to all who are affected. Adopt a versioning scheme for documents that clearly distinguishes drafts from baselined versions. A more robust solution is to store the requirements documents in a version control tool. Restrict read/write access to a few authorized individuals, but make the current versions available in read-only format to all project stakeholders. Even better, store your requirements in the database of a commercial requirements management tool. In addition to many other capabilities, such tools record the complete history of every change made in every requirement.

Keys to Excellent Software Requirements

While these ten traps aren’t the only ones lurking in the requirements minefield, they are among the most common and most severe. To avoid or control them, assemble a robust toolkit of practices for eliciting, analyzing, specifying, verifying, and managing a product’s requirements:

- Educating developers, managers, and customers about requirements engineering practices *and* the application domain
- Establishing a collaborative customer-developer partnership for requirements development and management
- Understanding the different kinds of requirements and classifying customer input into the appropriate categories
- Taking an iterative and incremental approach to requirements development
- Using standard templates for your vision and scope, use case, and SRS documents
- Holding formal and informal reviews of requirements documents
- Writing test cases against requirements
- Prioritizing requirements in some analytical fashion
- Instilling the team and customer discipline to handle requirements changes consistently and effectively

These approaches will help your next product's requirements provide a solid foundation for efficient construction and a successful rollout.

Bibliography

Carnegie Mellon University/Software Engineering Institute. *Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Mass.: Addison-Wesley, 1995.

Davis, Alan M. *Software Requirements: Objects, Functions, and States*. Englewood Cliffs, N.J.: PTR Prentice-Hall, 1993.

Gause, Donald C., and Gerald M. Weinberg. *Exploring Requirements: Quality Before Design*. New York: Dorset House Publishing, 1989.

IEEE Std. 830-1998, "Recommended Practice for Software Requirements Specifications." Los Alamitos, Ca.: IEEE Computer Society Press, 1998.

Leffingwell, Dean, and Don Widrig. *Managing Software Requirements*. Reading, Mass.: Addison Wesley Longman, 2000.

Sommerville, Ian, and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. New York: John Wiley & Sons, 1997.

Wiegers, Karl E. *Creating a Software Engineering Culture*. New York: Dorset House Publishing, 1996.

Wiegers, Karl E. *Software Requirements*. Redmond, Wash.: Microsoft Press, 1999.