

# In Search of Excellent Requirements<sup>1</sup>

**Karl E. Wieggers**

Process Impact  
www.processimpact.com

“I’ll go find out what they want, and the rest of you start coding.” This caption from a cartoon is uncomfortably close to the way some software organizations still treat the requirements specification process. Contemporary definitions of “quality” include the concepts of both meeting stated specifications and satisfying the actual customer requirements, which sadly are not always the same thing. Converging these two components into a unified vision of the final product is the linchpin of successful software development.

The problem of informal, incomplete, and poorly documented requirements is prevalent in small software groups, where a more casual development process lacks the rigor (and perhaps the rigidity) often found in larger organizations. As part of a multiyear process improvement effort [1, 2], our small software group focused on improved requirements specifications as the starting point for increasing our software quality and productivity.

Our quest for excellent requirements specifications has four components, which address the challenges of gathering, documenting, and validating user needs:

1. A high degree of customer participation in the development effort through a “project champion” model.
2. Preparation of structured specification documents in a format similar to the IEEE standard for software requirements specifications.
3. Construction of “dialog maps,” a variant of state transition diagrams that model complex user interfaces at a high level of abstraction.
4. Extensive use of prototypes to more fully elucidate user needs and explore alternative designs for satisfying them.

While these practices have not been universally applied on every project, we invariably obtain better results when we use them. Our experience indicates that these techniques can greatly reduce the expectation gap between the system the customer really needs to solve his problems and the system the developer decides to build for him.

## **OPTIMIZING CUSTOMER INVOLVEMENT**

Like many software organizations in large companies, our group develops products for internal corporate use only. The members of our software group are not experts in the specialized application domains that we support. Consequently, it is not reasonable to expect us to make sound business or technical decisions on behalf of the customers, or to resolve conflicting requirements supplied by different end users, or to set priorities for the many requirements that might be collected. On the other hand, we believe that customer involvement is the most critical factor in software quality, and that our most difficult problem is sharing the vision of the final product with the customer [3].

---

<sup>1</sup> Published in *The Journal of the Quality Assurance Institute*, January 1995. Reprinted with permission from the Quality Assurance Institute.

Our objective was to devise a mechanism by which the challenge of correctly defining a new system was shared by the developers and a small number of knowledgeable and energetic customer representatives. We wished to avoid the common scenario of requirements changing every day as yet another user heard about the program being built and threw in his two cents worth.

### **The Project Champion**

Since 1986, most of our projects have included one or more key customer representatives, or “project champions,” as integral members of the development team. The project champion is not a manager or a funding sponsor, but an actual end user of the system being constructed. The ideal champion is a respected member of the user community, eagerly anticipates delivery of the product, has a vision of what it should be, and has management support.

Our expectations of project champions are part of our written software development guidelines. The project champion is empowered to represent and make decisions on behalf of the entire customer community for the software system. For large projects or those with diverse user communities, a team of two to four co-champions may be appropriate. Each software project leader is responsible for negotiating the exact roles and responsibilities with his project champions.

The project champion serves as the primary interface between the customer community and the software developers. He is the focal point for collecting requirements from the customers for a new software system, and also for collecting and passing judgment upon defect reports and enhancement requests for existing systems. We expect the champion to provide substantial input to the requirements gathering process for the proposed system. He also assists with defining the testing and acceptance criteria that will be used to determine when the product is ready for delivery.

### **Project Champion Expectations**

In a nutshell, the project champion is responsible for supplying the correct specifications from the perspective of the user communities, and the software developer is responsible for satisfying these specifications. These are some of the specific activities we might ask champions to perform:

1. Work with the developers to determine the scope and limitations of the system and define the boundaries between the system and the rest of the world.
2. Solicit members of the customer community to provide input to the requirements for the new system and to participate in prototype evaluations or beta-testing.
3. Reconcile conflicting requirements so that a single unified set of specifications is presented to the developers. This may involve participation in Joint Application Design sessions or other group requirements gathering activities.
4. Distinguish between essential requirements and “chrome,” and establish implementation priorities accordingly.
5. Define security needs, performance requirements, and other software quality attributes, in addition to functional and user interface requirements.

6. Participate in inspections of requirements specifications documents, high-level designs, and system test plans. Assist with writing acceptance test criteria, developing test plans, and providing test datasets as appropriate.
7. Contribute to the writing of system documentation, including user manuals, on-line help displays, and training materials. The project champion usually is responsible for writing those parts of the documentation that pertain to the theoretical or conceptual foundation of the system.
8. Play a major advocacy role in “selling” the system to the customer or management communities. This is a recurrent problem with management-decreed software projects, which we have not yet learned how to solve.
9. Evaluate requests from users for correcting defects and adding enhancements. All such requests are entered into our change control system, which automatically sends the request by e-mail to the maintainer and the project champion. The champion determines priorities for these items and presents change requests to the maintainer.

### **How the Model Succeeds and Fails**

We have been fortunate to collaborate with some outstanding project champions. The best champions work hard to collect and evaluate diverse inputs to define the best possible system for the largest number of potential users. The project champion approach has not worked well on those few occasions when the champion felt that he had sole input into the system specifications, or, conversely, when he did not have a clear vision and strong opinions about what should be built.

Our best project champions have learned enough about our structured software development process to appreciate why we do things the way we do. At times, they have been valuable advocates on our behalf in the conflicts that inevitably arise between those programmers who want to build the product right the first time and those customers who always complain that it isn't done yet.

A major risk with the project champion model is that the champion's management will not appreciate the value of the contribution and therefore will not free up enough of the champion's time to do a proper job. It takes an exceptionally passionate and dedicated champion to put in project time above and beyond the normal job responsibilities. We are working to improve our customer management's appreciation of the importance of the project champion.

The champions must be empowered to make decisions about the software system, but they must never forget that the system is not being built for them alone. A good way to overcome excessive parochialism is to have a small cross-section of the user community supporting the champion. This backup team, including both regular and occasional users-to-be of the system, reviews specification documents and evaluates prototypes to provide some additional user perspectives and reality checks.

In every case where we have been forced to work on a management-mandated (as opposed to customer-driven) project and no champion was assigned, we have had serious problems in both determining and meeting customer needs. We have finally reached the state where if no project champion can be found to see that the right system is built, we cancel the project. The project champion model is so effective that there is no excuse for not following it when actual end users of the system are available locally to help build an excellent product.

## COMMUNICATION TOOLS

A wide assortment of techniques for gathering user requirements exists. A typical project might employ a combination of team meetings with project champions and developers, individual customer interviews, and user surveys. Regardless of the methods, the requirements must be organized, documented, and validated.

Most of the software products our group builds have a substantial user interface component. Effective methods are needed to represent and refine the user interface components of new software systems. User interface prototypes, dialog maps, and specification documents comprise a suite of communication tools that support our goal of having the software developers and their customers achieve a shared vision of the product being created.

### Software Requirements Specifications

Although we have written structured specification documents for many years, we have recently standardized on the IEEE Software Requirements Specification (SRS) template [4]. The IEEE SRS provides a well-structured framework for recording not only the functional requirements of a system, but also the many bits of ancillary information that are needed to do a quality job. Figure 1 illustrates the major sections specified in the IEEE SRS standard.

A particularly important section of the SRS is that dealing with software quality attributes (section 3.5). Many specification processes fail to investigate explicitly the importance of various quality attributes. The users and developers may both have thoughts about which of the “ilities” are important, but they are rarely recorded in the specification document.

Since many of the quality factors cannot be optimized simultaneously, it is essential to assess them for relative importance so that proper design decisions can be made [5]. The consequence of not explicitly discussing these quality tradeoffs is a surprise upon delivery, when the customer finds that his *implicit* quality attribute requirements have not been achieved. Surprises in software development are never good news; the goal of software engineering is to eliminate surprises late in the game.

We have begun incorporating quantitative and testable quality attributes in our SRS documents. For example, a reusable component that we expect to be enhanced frequently had a stated requirement for the maximum time needed to add a specific type of new feature. While specifications of quality attributes probably cannot be tested until the product is nearly complete, the act of documenting and studying such requirements can lead to design considerations that result in a better product, higher productivity, and fewer surprises.

The requirements specification process is prone to two pitfalls. At one extreme is the back-of-the-napkin spec that only crudely approximates the system you should be constructing to meet the user needs. The other extreme is the attempt to perfect the functional requirements and user interface components before implementing anything, also known as “analysis paralysis.”

One way to reach an appropriate middle ground in the specification process is to conduct formal inspections of the SRS. A structured document like the IEEE SRS is readily inspected by the design team, the project champions, other representative users, and other software engineers who are not directly involved with the project. We also use structured analysis methods and dataflow models to augment the narrative documents. It is well known that the cost of defect removal increases dramatically if the defect is found later in the development cycle, so we emphasize requirements inspection as a high-leverage quality practice.

## **Dialog Maps**

Our SRS documents often include mockups of the major user interface screens. As we are using windows extensively for even our mainframe applications, the number of screens and their possible interactions and navigation pathways has grown dramatically. Individual screen images scattered throughout a document provide a detailed representation of the contents and behavior of each display. However, we sought a technique that would provide an overall view of a complex user interface at a higher level of abstraction.

The dialog map is an efficient method for modeling a user interface. This tool is a modification of the state transition diagram (STD) [6]. A user interface resembles a finite state machine in that only one screen (state) is active at any given instant, and a set of defined navigation pathways and triggers (transitions) exists for moving from one screen to another.

As shown in Figure 2, each screen (rectangle) is identified only by name, with no detail at all shown about its fields or layout. The connections between one screen or window and another are shown as transition lines connecting the states. Dialog maps can be structured hierarchically to further control the degree of detail revealed at any specific level. We have used dialog maps to model systems containing over 100 screens and popup windows.

The dialog map helps to communicate the vision of the system architecture among all the participants: project champions, other users, developers, project managers, and customer managers. Common or similar interface features can be identified and built as reusable components. Any duplicated or missing functionality can be detected and corrected well before implementation.

Dialog maps can be drawn with CASE tools, which assist with validation of the STD and facilitate frequent iteration. Or they can be drawn with a pencil and paper. With either method, we have found great value in having this high-level view of the entire user interface. It is much easier to add a missing navigation pathway to a simple drawing than to connect two disparate functions in a completed system. Fixing such common faults early on helps avoid the blood-chilling customer response that begins with, "But I thought it was supposed to...."

## **Prototypes**

The user interface prototype is a powerful tool for both refining customer requirements and exploring design alternatives to satisfy those needs. A prototype is intended to answer specific questions about functionality or interaction styles. If you don't have any questions, don't bother with a prototype.

We try to put the minimum amount of effort into building and assessing a prototype that will provide good answers to our questions. If possible, use tools that allow you to construct a prototype quickly and modify it interactively. The effectiveness of prototyping is tied to the frequency and quality of iteration, as you sneak up on the best solution step by step. Electronic prototypes are not always required. Rettig provides some excellent suggestions about how to build and use paper prototypes with minimum effort and maximum impact [7].

It is tempting to perfect the prototype beyond the limited objective of answering your questions. This perfection usually adds little value unless you intend to evolve the prototype into the deliverable product. The decision to either evolve or abandon the prototype once it has served its initial purpose should be made explicitly.

Rather than simply letting some users play with the prototype and offer comments, we generally use structured prototype evaluation scripts intended to generate answers to our

specification or design questions. The scripts should be short and focused, so that the user is not daunted by the prospect of spending too much time working through them. In addition to many specific questions about the application, these are some general questions we have explored with prototype evaluation scripts:

- Does this program do the right thing for you?
- Do you find any defects or omissions in the user interface?
- Do you like the interaction style?
- Which of several example interaction techniques do you prefer?
- Do any parts of the interface feel clumsy, awkward, or inefficient?
- Will you enjoy using this program?

It is also informative to watch users work with the prototype. While few corporate software organizations can set up a full-blown usability testing lab, you can still learn a lot by actually seeing what the users try to do with your sample program. Where do their fingers move by instinct? When do they have that “you can’t get there from here” look of frustration? When must they write down information on paper so they can re-enter it on another screen? By watching users wrestle with a prototype, I have been able to make improvements that would not have been apparent from written or verbal responses to my prototype evaluation questions.

## **FROM REQUIREMENTS TO CODE**

Software engineers working in a small group do not always practice the formalities of software design. There is a tendency to segue from requirements directly into code, with perhaps a brief layover at architecture. This is particularly risky when significant time has passed since the requirements and user interface components were originally specified.

To maximize quality and minimize rework, it is wise to revisit the specs prior to implementation. No matter how carefully the specs are written and inspected, they always contain defects of omission and commission. Just before jumping into code, dust off your prototype and probe the depths of the small subset of the specifications you are preparing to implement.

A role-playing walkthrough of the specs with some users will help flesh out the details that no one can think of up front. Explore what-if scenarios, ponder the usability, and identify exception conditions that must be handled. Think about the context in which this particular function will operate, the different paths by which the user could arrive at the screen, and the various states the user’s session might be in when this feature is encountered. A last-minute walkthrough to update the requirements and the way you intend to implement them can go a long way toward reducing the expectation gap between initial concept and ultimate delivered reality.

## **CONCLUSIONS**

Software engineering is not just a computing business — it is also a communication business. The techniques described here focus on the critical communications interface between the users of a new software system and those who are building it. If you build the product that the customer initially tells you he wants, you will always wind up building it again, once you’ve finally understood the customer’s true needs. Even in a small software group, a focus on accurately and completely capturing, documenting, and modeling the user requirements is a major contributor to building high quality information systems.

**REFERENCES**

1. Wiegers, Karl E., "Implementing Software Engineering in a Small Software Group," *Computer Language*, June 1993, pp. 55-64.
2. Wiegers, Karl E., "Effective Quality Practices in a Small Software Group," *The Software QA Quarterly*, Spring 1994.
3. Wiegers, Karl, "Creating a Software Engineering Culture," *Software Development*, July 1994.
4. IEEE Std 830-1984, "Guide to Software Requirements Specifications," *IEEE Standards Collection: Software Engineering*. Piscataway, NJ: The Institute of Electrical and Electronics Engineers, 1993.
5. Deutsch, Michael S. and Willis, Ronald R., *Software Quality Engineering: A Total Technical and Management Approach*. Englewood Cliffs, NJ: Prentice Hall, 1988, Chapter 4.
6. Wasserman, A.I., "Extending State Transition Diagrams for the Specification of Human-Computer Interaction," *IEEE Transactions on Software Engineering*, **SE-11**, 8, (August 1985), pp. 699-713.
7. Rettig, Marc, "Prototyping for Tiny Fingers," *Communications of the ACM*, **37**, 4, (April 1994), pp. 21-27.

**Figure 1. IEEE Template for Software Requirements Specifications.**

Table of Contents

1. Introduction
  - 1.1 Purpose
  - 1.2 Scope
  - 1.3 Definitions, Acronyms, and Abbreviations
  - 1.4 References
  - 1.5 Overview
  
2. General Description
  - 2.1 Product Perspective
  - 2.2 Product Functions
  - 2.3 User Characteristics
  - 2.4 General Constraints
  - 2.5 Assumptions and Dependencies
  
3. Specific Requirements
  - 3.1 Functional Requirements
    - 3.1.1 Functional Requirement 1
      - 3.1.1.1 Introduction
      - 3.1.1.2 Inputs
      - 3.1.1.3 Processing
      - 3.1.1.4 Outputs
    - 3.1.2 Functional Requirement 2, etc.
  - 3.2 External Interface Requirements
    - 3.2.1 User Interfaces
    - 3.2.2 Hardware Interfaces
    - 3.2.3 Software Interfaces
    - 3.2.4 Communications Interfaces
  - 3.3 Performance Requirements
  - 3.4 Design Constraints
  - 3.5 Quality Attributes
  - 3.6 Other Requirements

Revision History

Figure 2. Sample dialog map.

