

Cover Story

Testing *in the* Dark

A pragmatic approach to overcoming undocumented requirements

by Johanna Rothman and Brian Lawrence

A project manager strides purposefully into your office. “This disk has the latest and greatest release of our software. Please test it. Today.” You say, “Okay, sure . . . what

does it do?” The manager stops in his tracks and says, “Uh, the usual stuff . . .”

Sound familiar? We’ve run into this situation as employees and as consultants. And

we’ve seen testers take the disk, stick it in the drive, and just start testing away.

That’s testing in the dark. We think there are approaches that are more productive. When we test or manage testers, we plan the testing tasks to know what value we can get from the testing part of the project.

Let’s try to take off the blindfold.

Even for a short (two-week) testing project, we’ve used this strategy. Consider this approach:

- Discover the product’s *requirements*, to know what testing needs to be done
- Define what *quality means to the project*, to know how much time and effort we can apply to testing
- Define a test plan, including *release criteria*, to check out different people’s understanding of what’s important about the product, and to know when we’re ready to ship

Discover the Requirements

The first part of planning is to play detective. Your product will have a variety of requirements over its lifetime, through several releases. Some

▶▶ QUICK LOOK

■ Discovering the requirements

■ Defining quality criteria

■ Success tips and key assumptions

requirements will be more important sooner, and others later. You have to discover *this* release's requirements.

At the beginning, you gather data; you cram your head full of a vast amount of relevant technical detail. In the next step, you transform the data into requirements specifics—you filter that information down into the form of requirements, where you really discover the product's intent.

Requirements are the reasons that drive design choices, and since you were handed a disk, the very fact that you have it in hand indicates that design decisions were made. The product was built based on its requirements; otherwise, you wouldn't have a product to test. Lots of people made lots of choices in building the software on that disk. The *reasons* for those choices are the requirements.

Software systems may have hundreds of requirements. You don't have to uncover them all—you get to choose how much you're willing to invest in finding out what the requirements are. Think about how much risk you want to take. The fewer requirements you probe in depth, the more risk you incur. You want the company to meet its release deadlines, but you're taking dangerous chances if you don't learn enough about the product's requirements to decide what to test and at what depth.

Customer requirements are the design decisions about your customer's *problem*. Use those design decisions to solve that problem with your product. A useful way to categorize customer requirements is to divide them into:

Users Users are people in roles, who often appear as the subjects of statements. *Who will the product affect? Directly, as end-users? Indirectly, by its very existence, and by others who are using it?*

Attributes Attributes are characteristics that appear as adjectives and adverbs in statements. *What characteristics do the users need? How reliable does your product have to be? How fast? What else?*

Functions Functions are actions

the system performs which appear as verbs in statements. *What does your product do for the users?*

Many people have different pieces of the picture. Programmers have been programming their bits. Marketers have been setting customers' expectations. Architects, managers, designers, and others have been discussing requirements in a variety of forums. Most often, their discussions focus on functions, rather than attributes. "Should we enable email here? How about making every Web link active?" Lots of assumptions have been made. While each contributor may know a great deal about his or her part, it's possible that no one has the complete picture. You need to be able to see that picture to test effectively.

You have to find out what got discussed in those forums. What decisions were made? *Did the requirements actually get implemented in the product?*

STEP 1: Gather Data

It doesn't matter how you get the information. What does matter is that you capture the product's design choices—the "goodies." You will use this information to design your tests and as input to decide what "quality" means for the product.

We have used a variety of ways to discover design choices:

■ **Read the documentation** Find the user documentation. Study it to see if you can understand what's going on. Look at the distribution media. Find the marketing material. Discover what the users will be expecting when they load up the product.

■ **Examine the product's architecture** Get someone who's supposed to understand the architecture to explain it to you. Get them to show you why they believe the product has the architecture they say it has. The architecture will often define the maximum upper limit of the product's capabilities. Find the weak links in the chain.

■ **Run the product** Examine its performance. Discover its reliability.

■ **Ask the developers** Ask the project leader which developers worked on which part of the project. When you know who

developed the code, ask questions about what the product does, and how the developer made those decisions. Have lunch with the relevant developer(s), and take notes. If you are intimidated because you don't feel as if you are a developer's "equal," get someone who *is* an equal to ask the questions for now. And an addition to your career goals may be in order: you want to acquire enough knowledge to be able to talk to developers as a peer.

■ **Ask the project manager** The project manager may have electronic or hardcopy documents you can review.

While you uncover the requirements, make sure you don't blame anybody. They did the best job they knew how at the time, and this is the result. As a detective, you have an advantage over the other people on the project. You may not be blinded by their assumptions. You have the opportunity to look at the product more objectively than they did, and compare what the product does with what it's supposed to do.

STEP 2: Transform "the Goodies" into Requirements

Now you've got some material to work with; it's time to convert it into requirements.

Let's suppose you've been questioning the developers of an embedded control system for a factory floor. Here is an example of the kinds of statements you get from the developers. The value of these statements will become clear once you boil them down and define what they all mean.

Provide five-day-per-week uptime.

Provide scheduling.

Restrict intruders' and guests' access.

Very easy to set up and customize.

Allow report modules.

Guide the user skillfully in the initial setup.

We want to identify the essence of each statement. Is the statement about users, attributes, or functions?

What does "Provide five-day-per-week uptime" tell you? This is about reliability. Someone is saying that their facility needs the

services that your software provides continuously during the workweek. Reliability is an attribute of this system.

How about “Restrict intruders’ and guests’ access”? The essence of this statement is security. Security is also an attribute of this system. “Intruders” and “guests” are potential users.

Go through the list, picking out the users, attributes, and functions to distill out the essence of your system. Organize your list by type, as users, attributes, and functions. Remember that this is a subset of what we would normally get because we are examining a subset of the feedback. Table 1 is a table of requirements based on the example above.

This is only a snapshot—not a complete picture of the requirements. And remember, these requirements are much more useful if they are specific enough to be testable.

For example, let’s look at reliability. One measure of reliability is Mean Time Between Failures (MTBF), which is the amount of time that a system runs before encountering a failure. There’s a big difference between 12 and 720 hours MTBF. That’s a working day versus a working month. Once you’ve determined the system’s desired MTBF, you have a specific and measurable target that you can test.

We want to derive a set of specific instances of each of the attributes, which will then serve as the basis for testing and release criteria. Once you have such a set, go back to the individuals you asked for information, and show them your set of requirements. Ask questions like “Does this set of requirements represent your understanding of what our software should be?” and “Is there anything misrepresented or missing from this set?”

Define What “Quality” Means to This Project

You’ve investigated the requirements, and you think you know *what* you have to test. Now you have to figure out how hard you need to test it.

The ultimate goal of a software

project is to add value to your company. Your testing goals must reflect *this* project’s quality criteria. Your product quality definition helps you plan how much testing you need to do in which areas.

Three possible goals (Robert Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992) for any software project might be:

- Minimize time to market, by delivering the release as soon as possible
- Maximize customer satisfaction, by delivering a rich set of features
- Minimize the number of defects

Consider choosing only one of these options. Once you know what drives the project, you can rank each requirement to form the basis for a prioritized test plan. For example, if you choose “time to market” as the critical goal, the ranked requirements might look like Table 2.

You can see that not all requirements are *critical* to your project’s success. You can choose how much to test each requirement based on how important each requirement is to your project.

As the in-the-dark tester, you look for the combination of requirements

that provides the critical set of features needed by your customer base.

Define a Test Plan Including Release Criteria

You know what you have to test, and you know what you’ve decided to deliver. Now you can define the test plan and the release criteria.

Consider each requirement in the context of the entire product, and plan what you will and will not test. You may want to develop use cases or scenarios to illustrate how to test each requirement or set of requirements. Specify the release criteria, so everyone knows what your specific testing goals are.

Define critical release criteria by first drafting them and then negotiating them with the project team. Two specific questions to ask about the product are:

- Must we meet this requirement by the requested release date?
- What is the effect on our customers if we do not meet this requirement by the release date?

Criteria for an initial release of the system in our factory control example

Table 1: Sample Requirements

TYPE	REQUIREMENTS SPECIFICS
Users	Operator – Person who administers the system
	Guest – Visitor, but not a normal user
	Intruder – Someone not authorized to use the system; this user is disfavored
Attributes	Reliable – The system is available; it never fails, 24 hours a day on weekdays
	Secure – Only authorized users may access this system
	Integrated – This system brings together different components
	Remotely accessible – There are ways to reach this system other than by the standard interface
Functions	Easy to install – Installing this system is simple and intuitive
	Feedback – Operators and users get indications about how their interactions with the system are working
	Inventory – Keep track of operating assets
	Report – Collate data and produce reports
	Schedule – Show future events
	Plan – Support selecting, trading off, and scheduling system events

might be:

- **Attribute: Reliability** Verify 5x24 uptime.
- **Function: Report** Produce reports over a one-week period.
- **User: Operator** functionality complete.

From your perspective, these may be reasonable criteria—if this is what you can test (verify) in the time you have. After you develop the release criteria, check with management to verify that you can test what they want to know about the product.

Defining release criteria shines a light onto assumptions and fears that you can now address in a controlled fashion.

Some Success Tips

Coming out of the dark entails defining what you need to do, and how you're going to do it. We've run into trouble when we've done this and have found these tips useful for success:

Clarify what management gets from their testing dollars and time

If you meet with project and/or senior management, and they want to know that the product will meet the 5x24 uptime requirement, you can explain what resources and time you need to do that testing. If you have ranked the requirements in your test plan, management can see where you are spending the time and when. Management can define their concerns, and you can address them. If you expect to release this product a number of times, you may choose to schedule post-release testing to understand the implementation of the other requirements.

The release criteria belong to the whole project

Make sure other people on the project review the release criteria. Confirm that you know what project quality is, and how that translates into non-negotiable ship criteria and project success.

Release the product based on the release criteria and no other criteria

Even if you think you now know everything about this project, test the criteria with the rest of the project staff and management. They will have different perspectives on the release criteria than you do, and interacting with them may help you think of needed changes. There is a point, however, beyond which no more tweaking can be allowed. At the end of the day, decide on a set of criteria and stick to it—then release the product based on that set of criteria and no other.

The Dark Side

While our approach to testing in the dark is highly effective in many cases, there are some circumstances where you're not likely to find our technique especially helpful. We are making some assumptions about your business setting, which, if not true, will limit your success.

ASSUMPTION 1: Testing in the dark is a workable alternative.

If you are working on high-reliability software systems, such as life-critical systems, you'd better not be in a test-in-the-dark situation. In critical system software, qualities such as reliability and security must be built in right from the start. We would expect a testing organization to help plan the development of such a system, and for all participants to have a good grasp on the system's requirements before design is begun. If you're in this kind of setting and you've been handed a complete program and told just to test it, what are the chances that your system will meet its critical requirements? Your testing may well be able to identify where your product fails to meet its requirements, but what are you going to do with that information? Fixing those kinds of problems probably demands a complete redesign, and it's probably too late for that.

ASSUMPTION 2: Communication is open.

Are you in a position where you can openly discuss your product's requirements? Will people tell you the

Table 2: Ranking Requirements

TYPE	REQUIREMENTS SPECIFICS	CRITICAL FOR TIME TO MARKET?
Users	Operator – Person who administers the system	Yes
	Guest – Visitor, but not a normal user	No
	Intruder – Someone not authorized to use the system; this user is disfavored	Yes
Attributes	Reliable – The system is available; it never fails, 24 hours a day on weekdays	Yes
	Secure – Only authorized users may access this system	Yes
	Integrated – This system brings together different components	No
	Remotely accessible – There are ways to reach this system other than by the standard interface	Yes
	Easy to install – Installing this system is simple and intuitive	No
Functions	Feedback – Operators and users get indications about how their interactions with the system are working	No
	Inventory – Keep track of operating assets	No
	Report – Collate data and produce reports	Yes
	Schedule – Show future events	No
	Plan – Support selecting, trading off, and scheduling system events	No

truth when you ask hard questions? Or will they just tell you something—anything—to make you go away? Just about everyone embraces the *idea* of open communication, but actually practicing it is another thing entirely. You have to evaluate your chances of getting genuine reasons for why your product was designed. If you come up empty—you just can't obtain reliable information—then there's little reason to base your testing on it. Our technique is best used in open environments, where people are not afraid to talk to each other. In a closed environment, trying this technique will just frustrate you.

**ASSUMPTION 3:
People actually want
to know the truth.**

This technique is about gathering the truth, so you may run into political opposition if you try it in a setting where hidden agendas rule the day. Some people *like* to keep things fuzzy and ambiguous, perhaps because they believe in the illusion that doing so gives them more choices and flexibility. Making plans, decisions, and criteria clear and unambiguous is the last thing these people want to see happen. They will oppose you, probably in subtle and devious ways. You have to decide if you want to play that kind of game. When you change the way people think about things, some gain and some lose power—and those who stand to lose won't like it. This technique challenges the status quo (albeit in a constructive way). While we recognize that in certain environments this may actually be hazardous to your continued employment, we don't think that's a problem—why would you want to work at a place that doesn't treat you like a professional?

Naturally we would prefer not to be in the test-in-the-dark situation at all. As testers, we're always better off participating in the design right from the start. But if you *are* in this position, and you're working on important but not life-critical software, and you have relatively good communications with other organizations, then we recommend this approach to help you choose what to test:

Diagnosing Risk

Jack Falk suggests having two business cards ready, just in case. “When you’re testing software with known requirements and reasonable lead time, you can be a Quality Assurance specialist,” he says, “but, honestly, there are times when your job title will really change to Risk Management.” Even with the best of intentions, he says, your team can’t build in quality in the last two weeks of the project.

Falk, co-author of *Testing Computer Software*, and now a part of the Quality Engineering team at Network Computer, Inc., remembers past jobs in which he had to switch between those job titles frequently. “It’s a little like being a doctor,” he recalls. “You go in saying ‘I’m sick, and it hurts when I do *this*...’ and the doctor’s job is to try to diagnose the type of problem, the severity, and tell you what kinds of treatment are possible—even when the news is bad.”

When you’re handed a completely “in the dark” project to assess two weeks before the release date, Falk warns, sometimes your most important job is to advise management on the risks the company is taking on. For example, do the financial considerations of releasing the product on schedule outweigh the *later* costs of increased customer support when the users run into problems? It’s better for everyone concerned if you’re very clear about what the company can and can’t do at that point, so you don’t become a silent partner in a risky release decision.

While he’s seen pre-release panics on new products, Falk says it’s even more common to see an undocumented project in the form of a product update that includes new features or bug fixes.

The differences between interim builds can be a headache, even in a place with good requirements habits. “If the builds are very similar,” he recalls, “and the developers aren’t good at communicating what’s different with this week’s version, you’ve got a challenge on your hands.”

Communication is the key, says Falk. If part of the QA person’s mission is to teach the other members of the organization the importance of sharing and documenting information, one of the best ways to teach is by example.

—A.W.

- Uncover the requirements
- Decide what quality means
- Define a plan and a set of release criteria so you can measure when you’ve met your plan

Lead yourself back into the light! **STQE**

Johanna Rothman speaks, writes, and consults on managing high-technology product development to improve productivity and quality. She has over twenty years of experience in software engineering and management. You can reach Johanna electronically at jr@jrothman.

com, www.jrothman.com, or by phone: 781-641-4046.

Brian Lawrence, of Coyote Valley Software, is an author and consultant with extensive experience in the software industry. He teaches and facilitates requirements analysis, peer reviews, project planning, risk management, life-cycles, and design specification techniques. He can be emailed at brian@coyotevalley.com. Both authors would like to thank James Bach, Dave Gelperin, Elisabeth Hendrickson, Bob Johnson, Pat Medvick, and Melora Svoboda for their valuable expertise and comments on testing in the dark.