What Do You Mean "There's Nothing to Test"?
How QA Can Help Keep Projects On Target Before Coding Starts

Date May 2007, Version 1

Author: Fred Hagan, Managing Test Lead, Quardev, Inc.

Oops, Wrong Tooth

Sure, everybody's had software development projects that released late and went over budget. So what's new. And in how many of these instances is the product not only late and over budget, but also a mere shadow of the product that was envisioned before the coding started? Imagine you're conducting a demo for the customer immediately after the code complete milestone is reached, and the customer says something like, "You've built a good application here. Unfortunately it's not the application we wanted." Bummer. So now you look like a dentist who pulled the wrong tooth. And pulled it late. And for too much money. Embarrassing is probably about as good as that situation gets. (And depending on where you work, it might not get that good twice.) So how did things get so far off track? Why did the wrong application get built? And most importantly, how could your team have avoided it?

Okay, Somebody Fess Up

Before we consider possible solutions, let's take a closer look at the problem and get the finger-pointing part out of the way. As you may know already from personal experience, even when the project manager, business analysts, and system architects meet regularly with the client to gather and refine requirements, the application that gets built can still differ significantly from the client's idea of the application that he/she is requesting (and paying for). So who's at fault when this happens—the development team or the client? The answer is usually Both and Neither.

Consider the following scenarios. Any of them sound familiar?

➡ The members of the development team and the customer, or end user, may come from different worlds, and may speak and understand different cultural tongues in accord with the point of view of their jobs (jargon). So it's possible that each party may think they all understand each other perfectly – until the product is actually created.

➡ Developers may try to make requirements fit an already partially existing solution rather than building a new solution specific to the requirements. Vague or extremely general requirements may make it easier for developers to be tempted into this mistake.

➡ The client may change his mind about what he wants, or may add new requirements late in the process, causing design and functional specs to change even after the coding has started (feature creep). This contributes to general confusion in the development process, and it may affect (break) other functionality or features, even ones that the changed code does not appear to touch. In this scenario, the programmers are trying to code to a moving target, and all bets are off.

➡️ Requirements might be "analyzed" by programmers or engineers who lack the domain knowledge to truly *get* the business need or the reason for certain requirements. In such cases, the development team just doesn't fully understand the problem that the requirement is trying to solve. Often, nobody even knows there's a misunderstanding until later. Requirements that are *implied* rather than explicitly stated may contribute to this problem.

Okay, you might be thinking, I've been around the block a few times and I've seen all those things happen on projects, but what does this have to do with testing? Excellent question. Let's start at the beginning.

The Role of QA Before Coding

Over the past few years, it's been apparent that the traditional quality assurance (QA) role in software development is changing. More and more, the view that QA is not merely an evaluative activity that comes at the very end of the project lifecycle seems to be taking hold. Steve McConnell's statement in *Rapid Development* that up to 70% of the defects in an application can be detected before any code is written is becoming widely known in the world of QA and development process models. And that's good news.

However, while many organizations may agree that QA should be involved very early on in the development lifecycle, it is often little more than lip service. Some may wonder exactly what it means to involve QA from the initial phases of a project, or how to go about doing it. Other than producing a very high-level and tentative draft of a test plan, what can QA really contribute before there's any code written? What do you test when it seems like there is nothing to test?

Requirements, Design, and Risk

Now, when testing an application or system, presumably the tests are based on the requirements and on the specifications that were created from those requirements during the design process. But what is sometimes forgotten is that the requirements and specification documents themselves are legitimate "test items," in the parlance of exploratory testing. James Bach, principal consultant at Satisfice, Inc. and one of the pioneers of Exploratory Testing as an actual discipline, has made the following observation: "Because of incompleteness and ambiguity [of requirements], testing should not be considered merely as an evaluative process. It is also a process of exploring the meaning and implications of requirements." (Bach, James. "Risk and Requirements Based Testing" in *Computer*, June 1999)

As Bach's statement suggests, the documented requirements of a system or application, whether documented explicitly or implied, are project artifacts worthy of QA scrutiny. In fact, the requirements as documented may, and probably do, have "bugs" that should be logged and tracked the same as bugs that will be reported later on against the actual code. Most projects would benefit from having requirements-phase and design-phase bugs as *major deliverables* from QA during the project's early stages. Why?

Well, William Perry, in his *Effective Methods for Software Testing,* offers a suggestion simply by stating the function of the requirements and design phases: "The requirements phase outputs define the customer's needs in a format that can be used to create design specifications…. Once verified as accurate and complete, the design can be moved to the build phase to create the code that will produce the needed results…." (Perry, p. 326). Conversely, one could say it follows that vague or ambiguous or incomplete requirements lead to design specifications which are "close but no cigar" when compared to what the client actually wanted, and which in turn lead to building a "close but no cigar" application or system.

Testing in the Pre-coding Phases

Okay, so how do you do it? Three things: Quality Criteria, Techniques, Deliverables. Any conversation about testing in the requirements and design phases should break out into these three topics. If the requirements and design document are artifacts to be tested before code, then what are the **quality criteria** that apply to them? On what basis are they evaluated as "good" or "not good," and with what sort of **test techniques** do we identify and articulate "bugs" to be reported against them? And finally, what QA artifacts do we create as **deliverables** or **work products** specific to the requirements definition and design phases of the project? For starters, check out the following table. It's just a place to begin.

Requirements and Design Phase Testing at a Glance:

Quality Criteria	Test Techniques or Actions	Possible Deliverables or Work Products
Testable – Unambiguous – Observable – Actionable	Design and write test cases – "Pass" cases – "Fail" cases	– Test cases – Bugs for untestable requirements – Traceability matrix
"Implied" requirements are clear – Unambiguous – Observable – Actionable	Walk-through with customer and business analyst	– Test cases to cover implied requirements – Bugs for untestable or ambiguous requirements – Bugs for missing requirements
Design conforms to requirements (Maps to specific business needs identified during requirements gathering.) (Perry, p.321)	Confirm mapping of design to requirements via walk-through with customer and business analyst.	– Discrepancies noted and itemized in general (narrative) report of QA findings in Requirements phase. – Bugs for design elements that do not map to requirement.
Design Facilitates use. (Perry, p.321)	Identify "use case" flows, including alternate and exception flows via walk-through with customer and business analyst.	Bugs for holes revealed in design for workflow and/or usability.

For documented requirements, the single most important quality criterion is that each requirement be "testable." That is, can a test case involving specific observable items or behavior, or specific actions by the user such as click steps or other inputs, be written for it? Is each requirement "unambiguous" with clear pass/fail criteria? Do any requirements "imply" other requirements that are not explicitly stated?

For each stated requirement, try to write at least one test case that's as specific and concrete as the requirement allows. At least one additional case to cover a "fail" scenario is usually also possible. *NOTE:* A test case that simply places the word "verify" in front of the stated requirement is not good enough. True, it may not be possible at that point to include specific steps if the interface has not yet been designed – you can't have a step to click the Go button if it hasn't yet been determined that there will even *be* a Go button – but it should be possible to include at least an initial state, an area of the application, a general action by the user, and an expected "pass" or "fail" result.

If writing a reasonable test case is not possible, then that's a bug against the requirement, as the requirement is untestable. Log it in your bug tracking system. The bug is resolved when the requirement is stated in a way that allows a specific test case to be designed and written for it. When a test case is successfully written, it should be mapped to the requirement in a traceability matrix document. Each test case should be traceable to at least one requirement and vice versa.

The act of writing the test cases for each requirement will often reveal "implied" requirements or specifications that should be noted by the QA lead, verified with the customer and the business analyst, and then communicated to the project manager (PM) and the designer. Although not technically bugs, it may be a good idea to document and track them in a bug database so that they will be taken into account and not forgotten about as the application is being designed.

Missing and "Implied" Requirements and Specifications

Implied requirements or specifications may be derived from competing (similar) products, older versions of the same product, customer comments during reviews or walk-throughs, or style guides for GUI design, to name a few sources. However, they may also be simply intuitive, drawing on what just "makes sense."

For example, a learning management system (LMS) for computer-based courseware *probably* should not allow students to have administrator rights. One can reasonably assume that a student should not be able to change exam scores, progress and performance reports, or other data that only an instructor or system administrator would normally have access to. Also, the student should probably not be able to access the work of other students on the network. If these kinds of considerations are not explicitly called out in the requirements or specification documents, that is obviously no indication that they should not be tested, and QA may legitimately log bugs against their absence in the documentation.

And so…

Logging bugs against the requirements documents or specifications may seem extreme to some, but as in the LMS example, it would help to safeguard against an unsecure and risk-laden system being released simply because these "obvious" requirements were not so obvious to the programmers and were not spelled out explicitly in the specifications to which they were writing their code.

Such bugs may be resolved and closed as soon as they are written into the requirements and specifications, which should take a very short time and be very inexpensive. Compare that time and expense to re-designing a portion of the product and rewriting who knows how much code farther downstream. That's pretty easy math. Still think testing is what happens at the END of the development lifecycle?