jama build great products™

TESTING THE REQUIREMENTS

> Someone once asked me when you can begin testing your software. "As soon as you've written your first requirement, you can begin testing," I replied.

It's hard to visualize how a system will function just by reading the requirements specification. Tests that are based on requirements help make the expected system behaviors more tangible to the project participants. And the simple act of designing tests will reveal many problems with the requirements long before you don't execute the tests on an operational system. If you begin to develop tests as soon as portions of the requirements stabilize, you can find problems while it's still possible to correct them inexpensively.

REQUIREMENT AND TESTS

Tests and requirements have a synergistic relationship. They represent complementary views of the system. Creating multiple views of a system—written requirements, diagrams, tests, prototypes, and so forth—gives you a much richer understanding of the system than can any single representation.

Agile software development methodologies often emphasize writing user acceptance tests in lieu of detailed functional requirements. Thinking about the system from a testing perspective is very valuable, but that approach still leaves you with just a single representation of requirements knowledge.

[1]Adapted from Karl E. Wiegers, **Testing the Requirements,** Microsoft Press, 2006.

© Jama Software, Inc

Writing black-box (functional) tests crystallizes your vision of how the system should behave under certain conditions. Vague and ambiguous requirements will jump out at you because you won't be able to describe the expected system response. When BAs, developers, and customers walk through the tests together, they'll achieve a shared vision of how the product will work.

A personal experience really brought home to me the importance of combining test thinking with requirements specification. I once asked my group's UNIX scripting guru, Charlie, to build a simple e-mail interface extension for a commercial defect-tracking system we were using. I wrote a dozen functional requirements that described how the e-mail interface should work. Charlie was thrilled. He'd written many scripts for people, but he'd never seen written requirements before. Unfortunately, I waited a couple of weeks before I wrote the tests for this e-mail function. Sure enough, I had made an error in one of the requirements. I found the mistake because my mental image of how I expected the function to work, represented in about twenty tests, was inconsistent with one of the requirement. Chagrined, I corrected the defective requirement before Charlie had completed his implementation, and when he delivered the script, it was defect free. It was a small victory, but small victories add up.

CONCEPTUAL TESTS

Of course, you cannot actually test your system when you're still at the requirements stage because you haven't written any executable software yet. Nonetheless, you can begin deriving conceptual tests from use cases or user stories very early on. You can then use the tests to evaluate functional requirements, analysis models, and prototypes. The tests should cover the normal flow of the use case, alternative flows, and the exceptions you identified during requirements elicitation and analysis.

As an illustration, consider an application called the Chemical Tracking System. One use case called "View Order" let the user retrieve a particular order for a chemical from the database and view its details. Some conceptual tests would be the following:

- User enters order number to view, order exists, user placed the order. Expected result: show order details.
- User enters order number to view, order doesn't exist. Expected result: Display message "Sorry, I can't find that order."
- User enters order number to view, order exists, user didn't place the order. Expected result: Display message "Sorry, that's not your order. You can't view it."

Ideally, a business analyst will write the functional requirements and a tester will write the tests from a common starting point, the user requirements, as shown in Figure 1. Ambiguities in the user requirements and differences of interpretation will lead to inconsistencies between the views represented by the functional requirements, models, and tests. Finding those inconsistencies reveals the errors. As developers gradually, translate the requirements into user interface and technical designs, testers can elaborate those early conceptual tests into detailed test procedures for formal system testing.

AN EXAMPLE

The notion of testing requirements might seem abstract to you at first. Let's see how the Chemical Tracking System team tied together requirements specification, analysis modeling, and early test-case generation. Following are a use case, some functional requirements, part of a dialog map, and a test, all of which relate to the task of requesting a chemical.

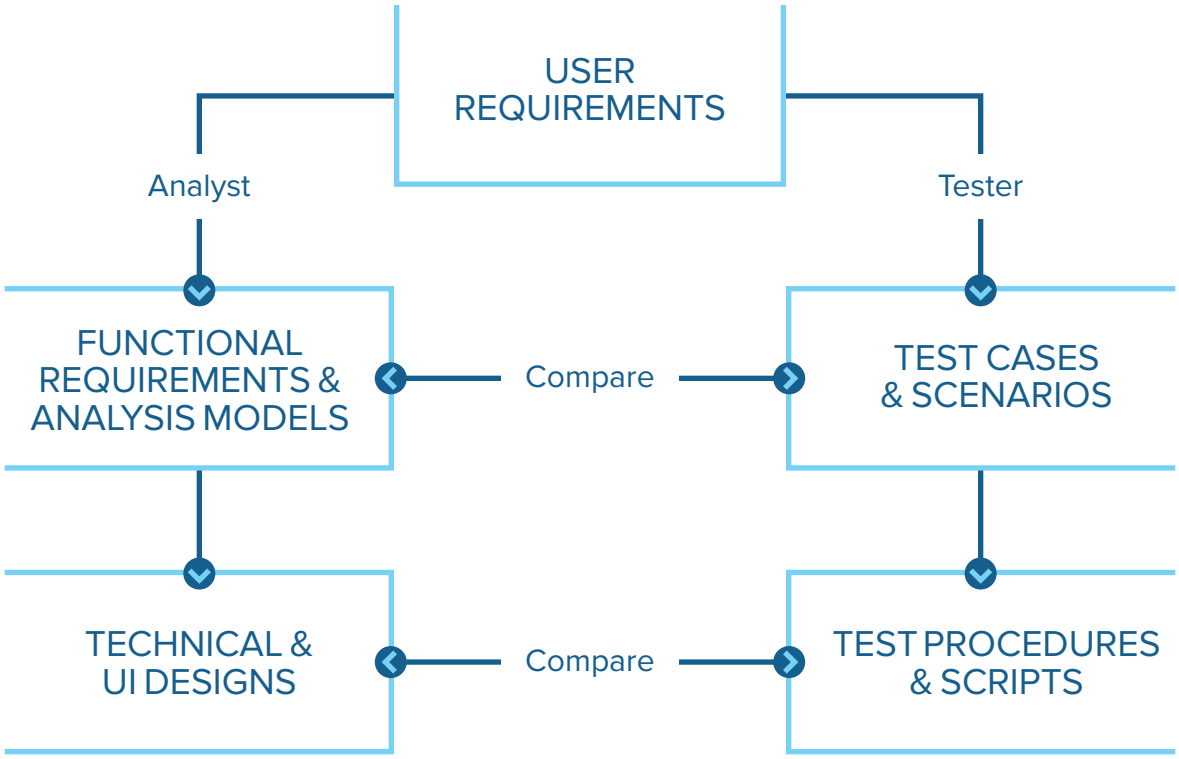

Figure 1. Development and testing work products derive from a common source.

Use Case.

A fundamental use case for the system is "Request a Chemical." This use case includes a path that permits the user to request a chemical container that's already available in the chemical stockroom. This option would help the company reduce costs by reusing containers we already have on hand instead of buying a new one. Here's the use case description:

The Requester specifies the desired chemical to request by entering its name or chemical ID number. The system either offers the Requester a new or used container of the chemical from the chemical stockroom, or it lets the Requester create a request to order a new container from a vendor.

Functional Requirement.

Here's a bit of functionality associated with this use case:

1. If the stockroom has containers of the chemical being requested, the system shall display a list of the available containers.

2. The user shall either select one of the displayed containers or ask to place an order for a new container from a vendor.

Dialog Map.

Figure 2 illustrates a portion of the dialog map for the "Request a Chemical" use case that pertains to this function. A dialog map is a high-level view of a user interface's architecture, modeled as a state-transition diagram. The boxes in this dialog map represent user interface displays (dialog boxes in this case), and the arrows represent possible navigation paths from one display to another.

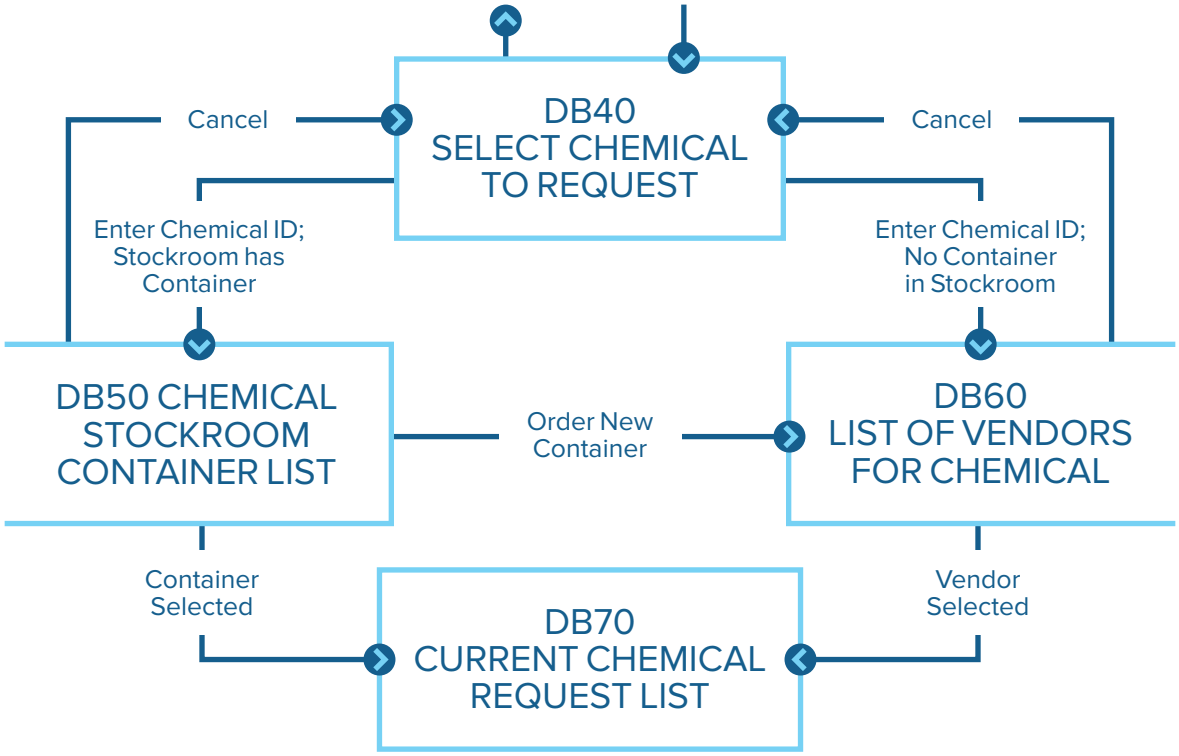

Figure 2. Portion of the dialog map for the "Request a Chemical" use case.

Test.

Because this use case has several possible execution paths, you can envision numerous tests to address the normal flow, alternative flows, and exceptions. The following is just one test, based on the path that shows the user the available containers in the chemical stockroom:

> At dialog box DB40, enter a valid chemical ID; the chemical stockroom has two containers of this chemical. Dialog box DB50 appears, showing the two containers. Select the second container. DB50 closes and container 2 is added to the bottom of the Current Chemical Request List in dialog box DB70.

Ramesh, the test lead for the Chemical Tracking System, wrote several tests like this one, based on his understanding of how the user might interact with the system to request a chemical. Such abstract tests are independent of implementation details. They don't describe clicking on particular buttons or other specific interaction techniques. As the user interface design activities progressed, Ramesh refined these abstract tests into specific test procedures.

Now comes the fun part—testing the requirements. Ramesh first mapped the tests against the functional requirements. He checked to make certain that each test could be executed by the existing set of requirements. He also made sure that at least one test covered every functional requirement. Such mapping usually reveals omissions. Next, Ramesh traced the execution path for every test on the dialog map with a highlighter pen. The yellow line in Figure 3 shows how the preceding sample test traces onto the dialog map.

By tracing the execution path for each test on the model, you can find incorrect or missing requirements, correct errors in the dialog map, and refine the tests. Suppose that after "executing" all the tests in this fashion, the navigation line in Figure 2 labeled "order new container" that goes from DB50 to DB60 hasn't been highlighted. There are two possible interpretations:

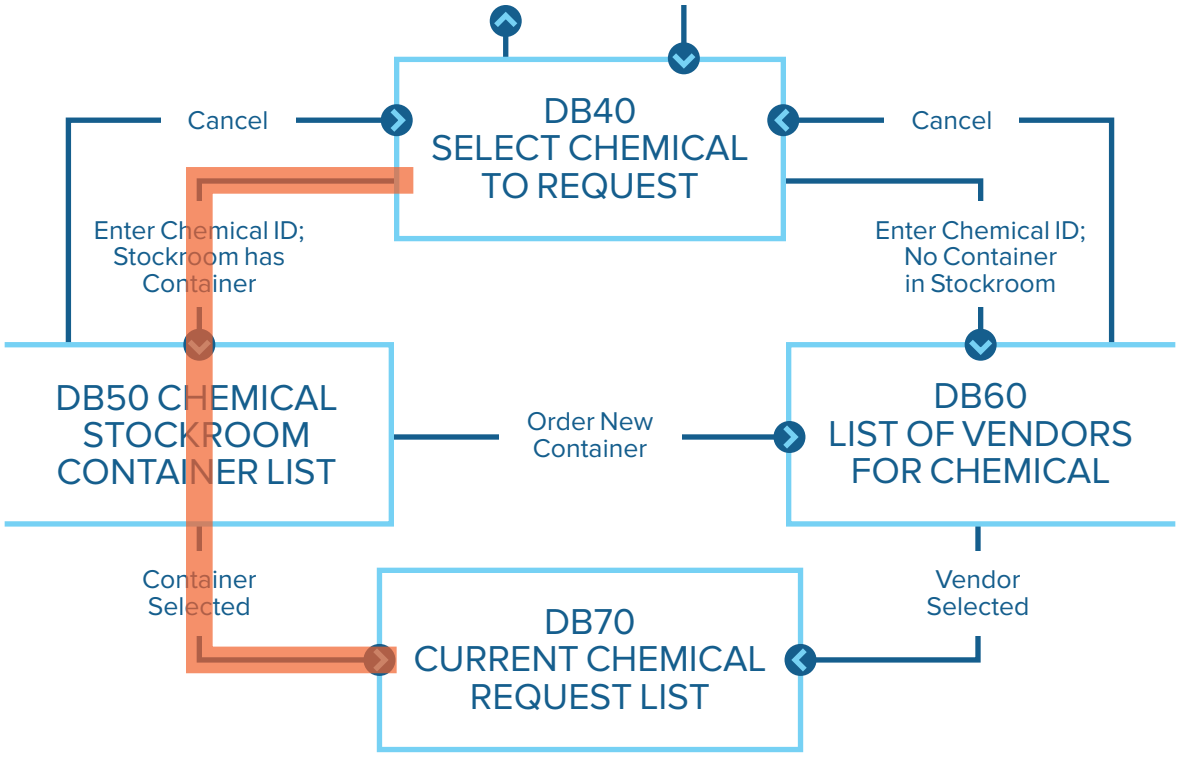

Figure 3. Tracing a test onto the dialog map for the "Request a Chemical" use case.

- The navigation from DB50 to DB60 is not a permitted system behavior. The BA needs to remove that line from the dialog map. If the SRS contains a requirement that specifies the transition, the BA must also remove that requirement.
- The navigation is a legitimate system behavior, but the test that demonstrates the behavior is missing.

When I find such a disconnect, I don't know which possible interpretation is correct. However, I do know that all of the views of the requirements—SRS, models, and test—must agree.

Suppose that another test states that the user can take some action to move directly from DB40 to DB70. However, the dialog map doesn't contain such a navigation line, so the test can't be "executed." Again, there are two possible interpretations:

- The navigation from DB40 to DB70 is not a permitted system behavior, so the test is wrong.
- The navigation from DB40 to DB70 is a legitimate function, but the dialog map and perhaps the SRS are missing the requirement that allows you to execute the test.

In these examples, the BA and the tester combined requirements, analysis models, and tests to detect missing, erroneous, or unnecessary requirements long before any code was written. Every time I use this technique, I find errors in all the items I'm comparing to each other.

As consultant Ross Collard pointed out, "Use cases and tests work well together in two ways: If the use cases for a system are complete, accurate, and clear, the process of deriving the tests is straightforward. And if the use cases are not in good shape, the attempt to derive tests will help to debug the use cases." I couldn't agree more. Conceptual testing of software requirements is a powerful technique for controlling a project's cost and schedule by finding requirement ambiguities and errors early in the game.

ABOUT KARL WIEGERS

Karl has provided training and consulting services worldwide on many aspects of software development, management and process improvement. He has authored 5 technical books, including Software Requirements, and written more than 175 articles. Prior to starting Process Impact in 1997, he spent 18 years at Eastman Kodak Company. His responsibilities there included experience as a photographic research scientist, software applications developer, software manager, and software process and quality improvement leader. Karl has led process improvement activities in small application development groups, Kodak's Internet development group, and a division of 500 software engineers developing embedded and host-based digital imaging software products. http://www.processimpact.com.

ABOUT JAMA SOFTWARE

From concept to launch, the Jama product delivery platform helps companies bring complex products to market. By involving every person invested in the organization's success, the Jama platform provides a structured collaboration environment, empowering everyone with instant and comprehensive insight into what they are building and why. Visionary organizations worldwide, including SpaceX, The Department of Defense, VW, Time Warner, GE, United Healthcare and Amazon.com use Jama to accelerate their R&D returns, out-innovate their competition and deliver business value. Jama is one of the fastest-growing enterprise software companies in the United States, having exceeded 100% growth in each of the past four years, during which time both Inc. and Forbes have repeatedly recognized the company as a model of responsible growth and innovation. For more information please visit http://www.jamasoftware.com.