CEM KANER ON SCENARIO TESTING

The Power of "What If..." and Nine Ways to Fuel Your Imagination

Once upon a time, a software company developed a desktop publishing program for the consumer market. During development, the testers found a bug: In a small zone near the upper right corner, you couldn't paste a graphic. They called this the "postage stamp bug." The programmers decided this wasn't a very important bug, since you could work around it by resizing the graphic or placing it a bit differently. The code was fragile, so they decided not to fix it.

The testers, on the other hand, felt the postage stamp bug should be fixed. To strengthen their case, they found someone who helped her children lay out their Girl Scout newsletter. The mother wanted to format the newsletter exactly like the one she had mimeographed, but she could not because the newsletter's logo was positioned at the postage stamp. The company still wouldn't fix the bug. The marketing manager said the customer only had to change the document slightly, and the programmers insisted the risk was too high.

Being a tenacious bunch, these testers didn't give up. The marketing manager often bragged that his program could do anything PageMaker could do, so the testers dug through PageMaker marketing materials and found a brochure with a graphic you-know-where. This bug report said the postage stamp bug made it impossible to duplicate PageMaker's advertisement. That got the marketer's attention. A week later, the bug was fixed.

This story (loosely based on real events) is a classic illustration of a *scenario* test. A scenario is a hypothetical story used to help a person think through a complex problem or system. Scenarios gained popularity in military planning in the United

INFO TO GO

- Good scenario tests are based on a story.
- Scenarios should be motivating, credible, complex, and easy to evaluate.

States in the 1950s. Scenario-based planning gained wide commercial popularity after a spectacular success at Royal Dutch/Shell in the early 1970s. (For some of the details, read *Scenarios: The Art of Strategic Conversation* by Kees van der Heijden, Royal Dutch/Shell's former head of scenario planning.) A *scenario test* is a test based on a scenario.

SCENARIO TEST APPLICATIONS

In the postage stamp example, the first report the testers filed came from a typical feature test. Everyone agreed there was a bug, but it didn't capture the imagination of any influential stakeholders. The second report told a credible story about a genuine member of the target market (the Girl Scout newsletter author), but that customer's inconvenience wasn't motivating enough to convince the marketing manager to override the programmers' concerns. The third report told a different story that limited the marketing manager's sales claims. That hit the marketing manager where it hurt. He insisted the bug be fixed.

The postage stamp bug illustrated *one* application of scenario testing: *Make a bug report more motivating.* There are several other applications, including learn the product, connect testing to documented requirements, expose failures to deliver desired benefits, explore expert use of the program, and bring requirements-related issues to the surface.

Early in testing, scenarios help you *learn the product.* I used to believe that an excellent way to teach testers about a product was to have them work through the manual, keystroke by keystroke. For years I did this myself and required my staff to do it. I was repeatedly confused and frustrated that I didn't learn much this way and annoyed with staff who treated the task as low value. Colleagues (James Bach, for example) have also told me they've been surprised that testing the product against the manual hasn't taught them much. John Carroll tackled this issue in his book *The Nurnberg Fun-*

9 WAYS TO CREATE GOOD SCENARIO TESTS

Because she has a different perspective, the scenario tester will often do her own product and marketing research while she tests, on top of or independently of research done by marketing. Here are some useful ways to guide your research. It might seem that you need to know a lot about the system to use these, and yes, the more you know, the more you can do. However, even if you're new to the system, paying attention to a few of these as you learn the system can help you design interesting scenarios.

1 Write life histories for objects in the system.

Imagine a program that manages life insurance policies. Someone applies for a policy. Is he insurable? Is he applying for himself or a policy on his wife, child, friend, competitor? Who is he *allowed* to insure? Why? Suppose you issue the policy. In the future he might pay late, borrow against the policy, change the beneficiary, threaten to (but not actually) cancel it, appear to (but not) die—much can happen. Eventually, the policy will terminate by paying out, expiring, or being canceled. You can write many stories to trace different start-to-finish histories of these policies. The system should be able to handle each story. (Thanks to Hans Schaefer for describing this approach to me.)

2 List possible users. Analyze their interests and objectives.

It's easy to say, "List all the possible users," but not so easy to list them. (Don Gause and Jerry Weinberg provide a useful brainstorming list in *Exploring Requirements,* p. 72.) Once you identify a user, try to imagine some of her interests. For example, think of a retailer's inventory control program. Users include warehouse staff, bookkeepers, store managers, salespeople, etc. Focus on the store manager. She wants to maximize store sales, minimize write-downs (explained below), and impress visiting executives by looking organized. These are examples of her interests. She will value the system if it furthers her interests.

Focus on one interest, such as minimizing write-downs. A store takes a write-down on an item when it reduces the item's value in its records. From there, the store might sell the item for much less, perhaps below original cost, or even give it away. If the manager's pay depends on store profits, write-downs shrink her pay. Some inventory systems can contrast sales patterns across the company's stores. An item that sells well in one store might sell poorly in another store. Both store managers have an interest in transferring that stock from the low-sale store to the high-sale one, but if they don't discover the trend soon enough, the sales season might be over (such as the Christmas season for games) before they can make the transfer. A slow system would show them missed opportunities, frustrating them instead of facilitating profit-enhancing transfers.

In thinking about the interest (minimize write-downs), we identified an objective the manager has for the system, something it can do for her. Her objective is to quickly discover differences in sales patterns across stores. From here, you look for features that serve that objective. Build tests that set up sales patterns (over several weeks) in different items at different stores, decide how the system should respond to them, and watch what it actually does. Note that under your analysis, it's an issue if the system misses clear patterns, even if all programmed features work as specified.

Don't forget disfavored users. As Gause and Weinberg point out, some users are disfavored. For example, consider an accounting system and an embezzling employee. This user's interest is to get more money. His objective is to use this system to steal the money. This is disfavored: the system should make this harder for the disfavored user rather than easier.

3 List "system events" and "special events."

An event is any occurrence that the system is designed to respond to. In *Mastering the Requirements Process,* Robertson and Robertson write about business events, such as placing an order. For any event, you'd like to understand its purpose, what the system is supposed to do with it, business rules associated with it, and so on. *(Continued on next page)*

nel: Designing Minimalist Instruction for Practical Computer Skill. People don't learn well by following checklists or material that is organized for them. They learn by doing tasks that require them to investigate the product for themselves. (Another particularly useful way to teach testers the product while developing early scenarios is to pair a subject-matter expert with an experienced tester and have them investigate together.)

Scenarios are also useful to *connect to documented software requirements*, especially requirements modeled with use cases. Within the Rational Unified Process, a scenario is an instantiation of a use case—take a specific path through the model, assigning specific values to each variable. More complex tests are built up by designing a test that runs through a series of use cases. (Ross Collard described use case scenarios in "Test Design: Developing test cases from use cases," *STQE,* July 1999; available at www.stickyminds.com.)

You can use scenarios to *expose failures to deliver desired benefits* whether or not your company creates use cases or other requirements documentation. The scenario is a story about someone trying to accomplish something with the product under test. In our example scenario, the user tried to create a newsletter that matched a mimeographed newsletter. The ability to create a newsletter that looks the way you want is a key benefit of a desktop publishing program. The ability to place a graphic on the page is a single feature you can combine with other features to obtain the benefit you want. A scenario test provides an end-to-end check on a benefit that the program is supposed to deliver. Tests of individual features and mechanical combination tests of related features or their input variables (using such techniques as combinatorial testing or orthogonal arrays) are not designed to provide this kind of check.

Scenarios are also useful for *exploring expert use of a program.* As Larry Constantine and Lucy Lockwood discuss in *Software for Use*, people use a program

(Continued from previous page) Special events are predictable but unusual occurrences that require special handling. For example, a billing system might do special things at year-end. The inventory system might treat transfers differently (record quantities but not other data) when special goods are brought in for clearance sales.

4 List benefits and create end-to-end tasks to check them.

What benefits is the system supposed to provide? If the current project is an upgrade, what benefits will the upgrade bring? Don't rely on only an official list of benefits. Ask stakeholders what they *think* the benefits of the system are supposed to be. Look for misunderstandings and conflicts among the stakeholders.

5 Interview users about famous challenges and failures of the old system.

Meet with users (and other stakeholders) individually and in groups. Ask them to describe the basic transactions they're involved with. Get them to draw diagrams and explain how things work. As they warm up, encourage them to tell you the system's funny stories, the crazy things people tried to do with the system. If you're building a replacement system, learn what happened with the predecessor. Along with the funny stories, collect stories of annoying failures and strange things people tried that the system couldn't handle gracefully. Later, you can sort out how "crazy" or "strange" these attempted uses of the system were. What you're fishing for are special cases that had memorable results but were probably not considered credible enough to mention to the requirements analyst. Hans Buwalda talks about these types of interviews (www.stickyminds.com).

6 Work alongside users to see how they work and what they do.

While designing a telephone operator's console (a specially designed phone), I traveled around the country watching operator/receptionists use their phones. Later, leading the phone company's test group, I visited customer sites to sit with them through training, watch them install beta versions of hardware and software, and watch ongoing use of the system. This provided invaluable data. Any time you can spend working with users learning how they do their work will give you ideas for scenarios.

7 Read about what systems like this are supposed to do.

So you're about to test an inventory management program and you've never used one before. Where should you look? Amazon lists 33 books with titles like *What to Look For in Warehouse Management System Software.* Google gives 26,100 hits for "inventory management system." This wealth of material documents user expectations, competitive issues, and common and uncommon scenarios for any type of business system.

8 Study complaints about the predecessor to this system or its competitors.

Software vendors usually create a database of customer complaints. Companies that write software for their own use often have an in-house help desk (user support) group that keeps records of user problems. Read the complaints. Take "user errors" seriously—they reflect ways that the users expected the system to work, or things they expected the system to do.

You might also find complaints about your product or similar ones online.

9 Create a mock business. Treat it as real and process its data.

Your goal in this style of testing is to simulate a real user of the product. For example, if you're testing a word processor, write documents—real ones that you need in your work.

Try to find time to simulate a business that would use this software heavily. Make the simulation realistic. Build your database one transaction at a time. Run reports and check them against your data. Run the special events. Read the newspaper and create situations in your company's workflow that happen to other companies of your kind. Be realistic, be demanding. Push the system as hard as you would push it if this really were your business. And complain loudly (write bug reports) if you can't do what you believe you should be able to do.

Not everyone is suited to this approach, but in the hands of a skilled tester, I saw this technique expose database corruptors, report miscalculators, and other compelling bugs under more complex conditions than we would have otherwise tested.

differently as they gain experience with it. Initial reactions to a program are important, but so is the stability of the program in the hands of the expert user. You may have months to test a moderately complex program. This time provides opportunity to develop expertise and simulations of expert use. During this period, one or more testers can develop full-blown applications of the software under test. For example, testers of a database manager might build a database or two. Over the months, they will add data, generate reports, and fix problems, all while gaining expertise themselves and pushing the database to handle ever more sophisticated tasks. Along the way, especially if you staff this work in a way that combines subject-matter expertise and testing skill, these testers will find credible, serious problems that would have been hard to find (hard to imagine the tests to search for them) any other reasonable way.

Scenarios are especially interesting for *surfacing requirements-related controversies*. Even if there is a signed-off requirements document, this only reflects the agreements that project stakeholders *have reached*. There are always ongoing disagreements. As Tom DeMarco and Tim Lister have often pointed out, ambiguities in requirements documents are often not accidental; they are a way of papering over disagreements. A project's requirements can change dramatically for reasons that are difficult to control early in the project:

- Key people on the project come and go. Newcomers bring new views.
- Stakeholders' level of influence changes over time.
- Some stakeholders don't grasp the implications of a product until they use it, and they won't (or can't) use it until it's developed enough to be useful. This is not unreasonable—in a company that makes and sells products, relatively few employees are chosen for their ability as designers or abstract thinkers.
- Some people whose opinions will become important aren't even invited to early analysis and design meetings. For example, to protect trade secrets, some resellers or key customers might be kept in the dark until late in the project.
- Finally, market conditions change, especially on a long project. Competitors bring out new products. So do makers of products that are to be interoperable with the product under development, and makers of products (I/O devices, operating system, etc.) that form the technical platform and environment for the product.

While surfacing controversies is important, it can lead to frustrating delays. A tester who suspects that a particular stakeholder might be unhappy with some aspect of the program creates a scenario test and shows the results to that stakeholder. By creating detailed examples of how the program works, or doesn't work, the scenario tester forces issue after issue. As a project manager, I've seen this done on my projects and have been frustrated and annoyed by it. Issues that I thought were settled were reopened at inconvenient times, sometimes resulting in unexpected late design changes. I had to remind myself that the testers didn't create these issues. With or without testing, genuine disagreements will have their effects: In-house stakeholders (such as salespeople or help desk staff) might support the product unenthusiastically; customers might be less willing to pay for it; end users might be less willing to adopt it. Scenario testers provide an early warning system for requirements problems that would otherwise haunt the project later.

CHARACTERISTICS OF GOOD SCENARIO TESTS

The ideal scenario test has five key characteristics. It is (a) a story that is (b) motivating, (c) credible, (d) complex, and (e) easy to evaluate. These aren't the only good characteristics a test can have. I describe several test techniques and their strengths in "What IS a Good Test Case?" (Find this paper at www.testingeducation.org/articles.) Another important characteristic is *power:* One test is more powerful than another if it's more likely to expose a bug. See the accompanying sidebar "Power-Up Scenarios" for more on how to increase the power of scenario tests.

The test should be *based on a story* about how the program is used, including information about the motivations of the people involved. Storytelling is an art. I don't know how to teach you to be a good storyteller. What I can do is suggest some things that might be useful to include in your stories and some ways to gather and develop the ideas and information that you'll include. (See the sidebar "9 Ways to Create Good Scenario Tests" for some of these ideas.)

The story should be *motivating*. A story is motivating if a stakeholder with influence wants the program to pass the test. (Anyone affected by a program is a stakeholder. A person who can influence development decisions is a stakeholder with influence.) A dry recital of steps to replicate a problem doesn't provide information that stirs emotions in people. To make the story more motivating, tell the reader *why* it is important, why the user is doing what she's doing, what she wants, and what the consequences of failure are to her. This type of information is normally abstracted out of a use case (see

POWER-UP SCENARIOS

Let's talk about *power* for a moment. A technique (scenario testing) focused on developing credible, motivating tests is not as likely to quickly bring to mind the extreme cases that power-focused techniques (such as stress, risk-based, and domain testing) are so good for. They are the straightest lines to failures, but the failures they find are often dismissed as unrealistic, too extreme to be of interest.

One way to increase a scenario's power is to exaggerate slightly. When someone in your story does something that sets a variable's value, make that value a bit more extreme. Make sequences of events more complicated, or add a few more people or documents. Hans Buwalda is a master of this. He calls these types of scenario tests "soap operas." (See "Soap Opera Testing" at www.stickyminds.com.)

Alistair Cockburn's excellent book *Writing Effective Use Cases,* p. 18, and John Carroll's discussion of the human issues missing in use cases in *Making Use: Scenario-Based Design of Human-Computer Interaction,* pp. 236–37.)

Along with impact on the user, a highly motivating bug report might consider the impact of failure on the user's business or on your own company. For example, a bug that only modestly impacts the users but causes them to flood your company with phone calls would probably be considered serious. A scenario that brings out such effects would be influential.

The ideal scenario test has a *credible* story. It not only *could* happen in the real world, stakeholders believe that something like it probably *would* happen. Sometimes you can establish credibility simply by referring to a requirements specification. In many projects, though, you won't have these specs, or they won't cover your situation. (Each approach discussed in the sidebar "9 Ways to Create Good Scenario Tests" is useful for creating credible tests.)

The story should involve a *complex use* of the program or a *complex environment* or a *complex set of data*. A *complex* story involves many features. You *can* create simplistic stories that involve only one feature, but why bother? Other techniques, such as domain testing, are easy to apply to single features and are more focused on developing power in these simple situations. The strength of the scenario is that it can help you discover problems in the relationships *among* the features.

Finally, the ideal scenario test is easy to evaluate—that is, it should be easy to tell whether the program passed or failed. Of course, every test result should be easy to evaluate. However, the more complex the test, the more likely that the tester will accept a plausible-looking result as correct. Ease of evaluation is valuable for all tests, but is especially important for scenarios because they are complex. Glen Myers discussed this in his classic *Art of Software Testing,* and I've seen other expensive examples of bugs exposed by a test but not recognized by the tester.

RISKS OF SCENARIO TESTING

I've seen three serious problems with scenario tests. One, other approaches are better for testing early, unstable code. The scenario test is complex, involving many features. If the first feature is broken, the rest of the test can't be run. Once that feature is fixed, the next broken feature blocks the test. In some companies, complex tests fail and fail all through the project, exposing one or two new bugs at a time. Discovery of some bugs has been delayed a long time until scenario-blocking bugs were cleared out of the way. To efficiently expose problems as soon as they appear, test each feature in isolation before testing scenarios.

Two, scenario tests are not designed for coverage of the entire program. It takes exceptional care to cover all the features or requirements in a set of scenario tests. Covering all the program's statements simply isn't achieved this way.

Finally, scenario tests are often heavily documented and used time and again. This seems efficient, given all the work it can take to create a good scenario. But scenario tests often expose design errors rather than coding errors. The second or third time around, you've learned what this test will teach you about the design. Scenarios are interesting tests for coding errors because they combine so many features and so much data. However, there are so many interesting combinations to test that I think it makes more sense to try different variations of the scenario instead of the same old test. You're less likely to find new bugs with combinations the program has already shown it can handle. Do regression testing with single-feature tests or unit tests, not scenarios.

Scenario testing isn't the only type of testing and should not be used exclusively. Scenario testing works best for complex transactions or events, for studying end-to-end delivery of the benefits of the program, for exploring how the program will work in the hands of an experienced user, and for developing more persuasive variations of bugs found using other approaches. STQE

DISTINGUISHING SCENARIO TESTS FROM REQUIREMENTS ANALYSIS

Designing scenario tests is much *like* doing a requirements analysis, but is *not* requirements analysis. They rely on similar information but use it differently.

- The requirements analyst tries to foster agreement about the system to be built. The tester exploits disagreements to predict problems with the system.
- The tester doesn't have to reach conclusions or make recommendations about how the product should work. Her task is to expose credible concerns to the stakeholders.
- The tester doesn't have to make the product design tradeoffs. She exposes the consequences of those tradeoffs, especially unanticipated or more serious consequences than expected.
- The tester doesn't have to respect prior agreements. (Caution: testers who belabor the wrong issues lose credibility.)
- The scenario tester's work need not be exhaustive, just useful.

Cem Kaner is Professor of Computer Sciences at Florida Tech. He is senior author of three books, Lessons Learned in Software Testing, Bad Software, *and* Testing Computer Software. *He's also an attorney (a former prosecutor) whose idea of a good time is holding companies accountable for releasing defective software.* *Work towards this article was supported by the National Science Foundation grant EIA-0113539 and by Rational Software.*

***STQE* magazine is produced by Software Quality Engineering.**