An Introduction to Scenario Testing

Cem Kaner, J.D., Ph.D.

Florida Institute of Technology, April 2013

This is an updated version of a paper initially published in Software Testing & Quality Engineering magazine (2003). The underlying research was partially supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

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 very important. 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 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 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 States in the 1950's. Scenario-based planning gained wide commercial popularity after a spectacular success at Royal Dutch/Shell in the early 1970's. (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.

I think the ideal scenario test has several characteristics:

- The test is *based on a story* about how the program is used, including information about the motivations of the people involved.
- The story is *motivating*. A stakeholder with influence would push to fix a program that failed this test. (Anyone affected by a program is a stakeholder. A person who can influence development decisions is a stakeholder with influence.)
- The story is *credible*. It not only *could* happen in the real world; stakeholders would believe that something like it probably *will* happen.
- The story involves a *complex use* of the program or a *complex environment* or a *complex set of data.*
- The test results are *easy to evaluate*. This is valuable for all tests, but is especially important for scenarios because they are complex.

The first postage-stamp report 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, 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.

Why Use Scenario Tests?

The postage stamp bug illustrated *one* application of scenario testing: *Make a bug report more motivating*.

There are several other applications, including these:

- Learn the product
- Connect testing to documented requirements
- Expose failures to deliver desired benefits
- Explore expert use of the program
- Bring requirements-related issues to the surface, which might involve reopening old requirements discussions (with new data) or surfacing not-yet-identified requirements.

Early in testing, *use scenarios to 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 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 Funnel: 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 "Developing test cases from use cases." For a more comprehensive treatment, see Alistair Cockburn's, *Writing Effective Use Cases.*

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 her 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 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 their book, *Software for Use*, people use the program differently as they gain experience with it. Initial reactions to the 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, fix problems, 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 reflects the agreements that project stakeholders *have reached*. But there are also ongoing disagreements. As Tom DeMarco and Tim Lister point out, ambiguities in requirements documents are often not accidental; they are a way of papering over disagreements ("Both Sides Always Lose: Litigation of Software-Intensive Contracts", *Cutter IT Journal*, Volume XI, No. 4; April 1998).

A project's requirements can also 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 change 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 far enough developed 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 opinion 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.

A tester who suspects that a particular stakeholder would 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 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. 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 Scenarios

A 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?". Another important characteristic is *power*: One test is more powerful than another if it's more likely to expose a bug. I'll have more to say about power later. For now, let's consider the criteria that I describe as the strengths of scenario tests.

Writing a scenario involves writing a *story*. That's 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.

A scenario test is *motivating* if a stakeholder with influence wants the program to pass the test. 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 are the consequences of failure to her. This type of information is normally abstracted *out* of a use case (see 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*, p. 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 company (the software developer). For

example, a bug that only modestly impacts the user 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.

A scenario is *credible* if a stakeholder with influence believes it will probably 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 below is useful for creating credible tests.

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, easy to apply to single features and 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.

This brings us to *power.* A technique (scenario testing) focused on developing credible, motivating tests is not as likely to bring quickly 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; 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.)

The final characteristic that I describe for scenario tests is ease of evaluation—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. 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.

Context Analysis for Scenario Test Design

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.

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.

Understanding the human system in which a piece of software will operate is essential for designing scenario tests that are coherent, credible, compelling, appropriately complex, and easy-to-evaluate. To help with develop such understanding, testers sometimes study systems theory (Laszlo, 1996; Von Bertalanffy, 1976; Weinberg, 1991, 2001). Barab, Evans, and Baek (2004) have proposed that "activity theory might provide an ideal position–one with sufficient scope and depth–for observing individuals at work, alone or in collaboration with others, using electronic tools" (p. 205). Cultural Historical Activity Theory (Engeström, 1987), which we will often abbreviate as *CHAT* or *Activity Theory,* is an appropriate framework for examining the context in which software is used. Rebecca Fiedler and I have previously written about CHAT for the testing community in 2007 and 2009.

CHAT (Engeström, 1987) includes the collection of actors, actions, artifacts, and related activities that surround human endeavors. The model provides a framework for systems thinking that brings out the context in which a human activity is done such as the context(s) in which software is developed and used. CHAT has its roots in the Soviet psychological tradition–particularly work by Vygotsky, Leontev, and Luria (Barab, Evans, & Baek, 2004) and it has been repeatedly applied to software development in human-computer interaction and computer-supported cooperative work (Halverson, 2002; Kuutti, 1995, 1996; Kuutti & Arvonen, 1992; Nardi & Engestrom, 1998; Nardi & Redmiles, 2002), especially as these fields have begun to shift from user-centered design to context-centered design (Gay & Hembrooke, 2004; Kaptelinin & Nardi, 2006). When we talk about context-driven software testing (Kaner, Bach, & Pettichord, 2001 and the context-driven testing site, (www.contextdriventesting.com), the types of information that we consider "context" are the types of information that CHAT helps us identify and evaluate.

The CHAT framework is represented by a series of embedded triangles. See figure 1. In general terms, the three sides of the largest triangle represent a subject acting on an object while embedded in a cultural community. The nodes suggest interactions between the various components. For example, the subject's action upon the object is mediated by tools. Similarly, interactions between a subject and the broader community are governed by rules, norms, and customs. Finally, the community interacts with objects through division of labor. All of these actions and interactions are motivated by a specific intended outcome.

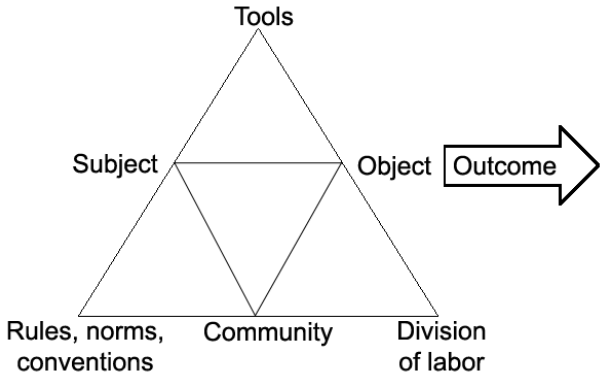

Figure 1: Cultural historical activity theory model

Seventeen Ways to Create Good Scenarios

Here are some useful ways to guide your analysis of how software will be used. 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.

Seventeen Ways to Create Good Scenarios
1. Create follow-up tests for bugs that look controversial or deferrable.
2. List possible users, analyze their interests and objectives.
3. Work alongside users to see how they work and what they do.
4. Interview users about famous challenges and failures of the old system.
5. Transactions
6. Sequences of work
7. Consider disfavored users: how do they want to abuse your system?
8. Forms
9. Write life histories for objects in the system
10. List "system events." How does the system handle them?
11. List "special events" or "calendar events." What accommodations does the system make for these?
12. List benefits and create end-to-end tasks to check them.
13. Competing systems
14. Study complaints about the predecessor to this system or its competitors.
15. Create a mock business. Treat it as real and process its data.
16. Try converting real-life data from a competing or predecessor application.
17. Output (reports/displays/Export files)

1. Create follow-up tests for bugs that look controversial or deferrable.

The postage stamp bug illustrates one type of scenario test but it is NOT the important kind of scenario. When testers think a bug is important to fix, a persuasive scenario can help them advocate for that work but this is just one special case of scenario testing.

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*, page 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 writedowns (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 interests, such as minimizing writedowns. A store takes a writedown 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, writedowns 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 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 Xmas 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 writedowns), 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.

3. 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. In particular, look at what they are doing and try to understand why. Identify things that irritate or confuse them and use that information to shape your scenario tests.

4. 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 "strange" or "crazy" 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).

5. Transactions

In a "Transaction processing" system, the transactions are indivisible operations. The system either completes the transaction or cancels it. For example, in a banking system, you can't half-open a bank account. You either open it or you don't. Identify all the possible transactions in a system and design scenarios for them. String several transactions together to develop scenarios for larger tasks composed of several transactions.

6. Sequences of work

People or the systems they use typically do tasks in some kind of order. What are the most common sequences of subtasks to complete the larger tasks. Elisabeth Hendrickson often analyzes programs by creating sequence diagrams of the part of the system she's working with at the moment. With sequence diagrams, you would normally create use-case level tests, and she often does that. But any time it's useful, she adds the human details that transform these into scenarios.

7. Consider disfavored users: how do they want to abuse your system?

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.

8. Forms

Many businesses use paper forms. Employees or customers fill out the forms. As their computer systems evolve, they let people enter data directly into online copies of the forms. Every form can be the source of many scenarios.

9. 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—lots can happen. Eventually, the policy will terminate by paying out or expiring or being cancelled. 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.)

Just as you can create a list of possible users and base your scenarios on who they are and what they do with the system, you can create a list of objects and base your scenarios on what they are, why they're used, and what the system can do with them.

10. List "system events." How does the system handle them?

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, events that have

meaning to the business, such as placing an order for a book or applying for an insurance policy. As another example, in a real-time system, anything that generates an interrupt is an event. 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. Robertson and Robertson make several suggestions for finding out this kind of information.

11. List "special events." What accommodations does the system make for these?

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

12. 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 only on 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.

13. Competing systems

You can learn a lot about a program you are testing by studying its competitors or reading about what systems like this are supposed to do. Where should you look if you're about to test an inventory management program and you've never used one before? I just checked Amazon and found 33 books with titles like *What To Look For In Warehouse Management System Software*, and *Quick Response: Managing the Supply Chain to Meet Consumer Demand*. Google gave 26,100 hits for "inventory management system." There's a wealth of material for any type of business system, documenting user expectations, common and uncommon scenarios, competitive issues and so on.

If subject matter experts are unavailable, you can learn much on your own about the business processes, consumer products, medical diagnostic methods or whatever your software automates. You just have to spend the time.

Anything that attracts your interest in another program is a potential scenario test in yours.

14. 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 will probably also find complaints about your product or similar ones online.

15. 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 I've seen it used with superb effect. In the hands of one skilled tester, this technique exposed database corruptors, report miscalculators, and many other compelling bugs that showed up under more complex conditions than we would have otherwise tested.

16. Try converting real-life data from a competing or predecessor application.

Running existing data (your data or data from customers) through your new system is a time-honored technique.

A benefit of this approach is that the data include special cases, allowances for exceptional events, and other oddities that develop over a few years of use and abuse of a system.

A big risk of this approach is that output can look plausible but be wrong. Unless you check the results very carefully, the test will expose bugs that you simply don't notice. According to Glen Myers, *The Art of Software Testing*, 35% of the bugs that IBM found in the field had been exposed by tests but not recognized as bugs by the testers. Many of them came from this type of testing.

17. Output (reports/displays/Export files)

Many pieces of software process data and produce reports or files with the results. Look to competing products, regulatory requirements, and business needs for ideas about the types of output your software should generate. Design scenario tests around the reporting, display, and file sharing needs you identify in your analysis.

Risks of Scenario Testing

I've seen three serious problems with scenario tests:

- 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. Test each feature in isolation before testing scenarios, to efficiently expose problems as soon as they appear.
- Scenario tests are not designed for coverage of the 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.

- 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.

In Sum

Scenario testing isn't the only type of testing. For notes on other types of tests that you might use in combination with scenario testing, see my paper, What IS a Good Test Case).

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.

References

Barab, S. A., Evans, M. A., & Baek, E. (2004). Activity theory as a lens for characterizing the participatory unit. In D. Jonassen (Ed.), *Handbook of research for educational communications and technology* (2nd ed., pp. 199-214). Lawrence Erlbaum Associates.

Buwalda, H, (2004, February), "Soap Opera Testing." Better Software magazine. http://logigear.com/resources/articles-presentations-templates/246-soap-opera-testing.html

Carroll, J.M. (1998), The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill. MIT Press.

Carroll, J.M. (2003), Making Use: Scenario-Based Design of Human-Computer Interactions. MIT Press.

Cockburn, A. (2000), Writing Effective Use Cases. Addison-Wesley.

Collard, R. (1999, July), "Developing test cases from use cases". Software Testing & Quality Engineering magazine. http://www.stickyminds.com/

Constantine, L.L. and Lockwood, L.A.D. (1999), Software for Use. Addison-Wesley.

Engeström, Y. (1987). Learning by Expanding: An Activity-Theoretical Approach to Developmental Research. Retrieved February 12, 2005, from http://lchc.ucsd.edu/MCA/Paper/Engestrom/expanding/toc.htm

Fiedler, R.L. & Kaner, C. (2009), "Putting the context in context-driven testing (an application of Cultural Historical Activity Theory)". Conference of the Association for Software Testing. Colorado Springs, CO. http://www.kaner.com/pdfs/FiedlerKanerCast2009.pdf

Gause, D. C., & Weinberg, G. M. (1989). Exploring Requirements: Quality Before Design. New York: Dorset House.

Gay, G., & Hembrooke, H. (2004). Activity-Centered Design: An Ecological Approach to Designing Smart Tools and Usable Systems. Cambridge, MA: The MIT Press.

Halverson, C. A. (2002). Activity theory and distributed cognition: Or what does CSCW need to do with theories? Computer Supported Cooperative Work (CSCW), 11(1), 243-267.

Kaner, C. (2007), "What is a good test case?" Software Testing Analysis & Review Conference (STAR) East, Orlando, FL, May 12-16, 2003. http://www.kaner.com/pdfs/GoodTest.pdf

Kaner, C. (2007), "I speak for the user: The problem of agency in software development," Quality Software & Testing Magazine. http://www.kaner.com/pdfs/I%20Speak%20for%20the%20User.pdf

Kaner, C., Bach, J., & Pettichord, B. (2001). Lessons Learned in Software Testing: Wiley.

Kaptelinin, V., & Nardi, B. A. (2006). Acting With Technology: Activity Theory and Interaction Design. Cambridge, MA: The MIT Press.

Kuutti, K. (1995). Work processes: Scenarios as a preliminary vocabulary. In J. M. Carroll (Ed.), Scenario-Based Design: Envisioning Work and Technology in System Development (pp. 19-36). New York: John Wiley & Sons.

Kuutti, K. (1996). Activity theory as a potential framework for human-computer interaction research. In B. A. Nardi (Ed.), Context and Consciousness: Activity Theory and Human-Computer Interaction. Cambridge, MA: MIT Press.

Kuutti, K., & Arvonen, T. (1992, October 31-November 4). Identifying potential CSCW applications by means of activity theory concepts: A case example. Paper presented at the CSCW'92 Conference on Computer-Supported Cooperative Work, Toronto.

Laszlo, E. (1996). The Systems View of the World: A Holistic Vision for Our Time (Advances in Systems Theory, Complexity, and the Human Sciences). Cresskill, NJ: Hampton Press.

Nardi, B. A., & Engestrom, Y. (1998). A Web on the Wind: The Structure of Invisible Work: Special Issue of Computer Supported Cooperative Work. Computer Supported Cooperative Work, 8(1-2).

Nardi, B. A., & Redmiles, D. F. (2002). Activity Theory and the Practice of Design: Special Issue of Computer Supported Cooperative Work. Computer-Supported Cooperative Work, 11(1-2).

Myers, G. (1979), The Art of Software Testing. Wiley.

Robertson, S. & Robertson, J. (2012, 3rd Ed.), Mastering the Requirements Process: Getting Requirements Right. Addison-Wesley.

van der Heijden, K. (2005, 2nd Ed.), Scenarios: The Art of Strategic Conversation. Wiley.

Von Bertalanffy, L. (1976). General System Theory: Foundations, Development, Applications. New York: George Braziller.

Weinberg, G. M. (1991). Quality Software Management: Systems Thinking (Vol. 1): Dorset House.

Weinberg, G. M. (1993). Quality Software Management: First-Order Measurement (Vol. 2): Dorset House.

Weinberg, G. M. (2001). An Introduction to General Systems Thinking (Silver Anniversary Edition). New York: Dorset House.