Acquisition Archetypes

Changing Counterproductive Behaviors in Real Acquisitions

"Happy Path" Testing

Robustness (that is, "negative") testing is a standard part of any comprehensive testing approach. It attempts to stress the system by providing "bad" or invalid inputs that the system should either reject, or tolerate gracefully. In this Acquisition Archetype, we look at a project where a development team found itself in crunch mode, and robustness testing took a back seat. Testing instead followed the "happy path"—a tightly scripted process that didn't duplicate real-world conditions, and only verified that the required functionality was in place and functioning correctly.

Starting Down the Path

The team's tests of the system did not represent actual operations, because the team did not test real interactions in a realistic environment. Instead, testing followed the "happy path," verifying that the system came up with the right answers—given the right inputs.

Later, in a review of the project, a government user said that the testing "was all scripted."

"We brought up what happens when everything isn't right. But the contractor didn't encourage that kind of testing."

"All of the system test scripts ran fine, but they weren't real world tests," the user said. "A couple of us took the privilege of deviating from the scripts, to test [the system] more thoroughly, and see if it would blow up."

The official test scripts, government users said, always delivered the correct end result. They found that disturbing. "We brought up issues of what happens when everything isn't right," said one. "But the contractor didn't encourage that kind of testing. [The contractor] was adamant that that wasn't what their testing was for."

Missing Your Defects and Finding Them Again

None of the initial testing revealed performance as a problem. It became a major issue at the next stage of the program, with a pilot at a single site (the full implementation would eventually encompass many more sites). This real-life test presented a slice of the actual working environment.

"In [initial] testing, they never had problems," a user said, "and transactions went really fast—they said, 'Wow!' it was so fast. When things went live [at the single site], all of the problems started—it was a world of difference in the real performance versus the scripted test performance."

Another user said the problems they ran into with the single-site pilot occurred because "all of the development work and initial testing was done in the 'city of Perfect' … everything worked perfectly. The problem was with the [real life input] errors and discrepancies, and that caused the problems."

Robustness testing, and its attempts to break the system by using bad inputs, likely would have revealed the flaws.

Rework… and More Schedule Pressure

The contractor help desk was swamped by the trouble reports that poured in from users in the single site deployment. Developers had to be pulled off new tasks to fix the problems—to do rework.

"All of the system test scripts ran fine, but they weren't real world tests."

Putting developers on bug fixing, of course, leads directly to a worsening schedule crunch. All of the rework wasn't planned for, and the program didn't have adequate resources. The team had traveled the Happy Path—but found it was anything but a shortcut. In the end, no one was happy.

(Continued on page 2)

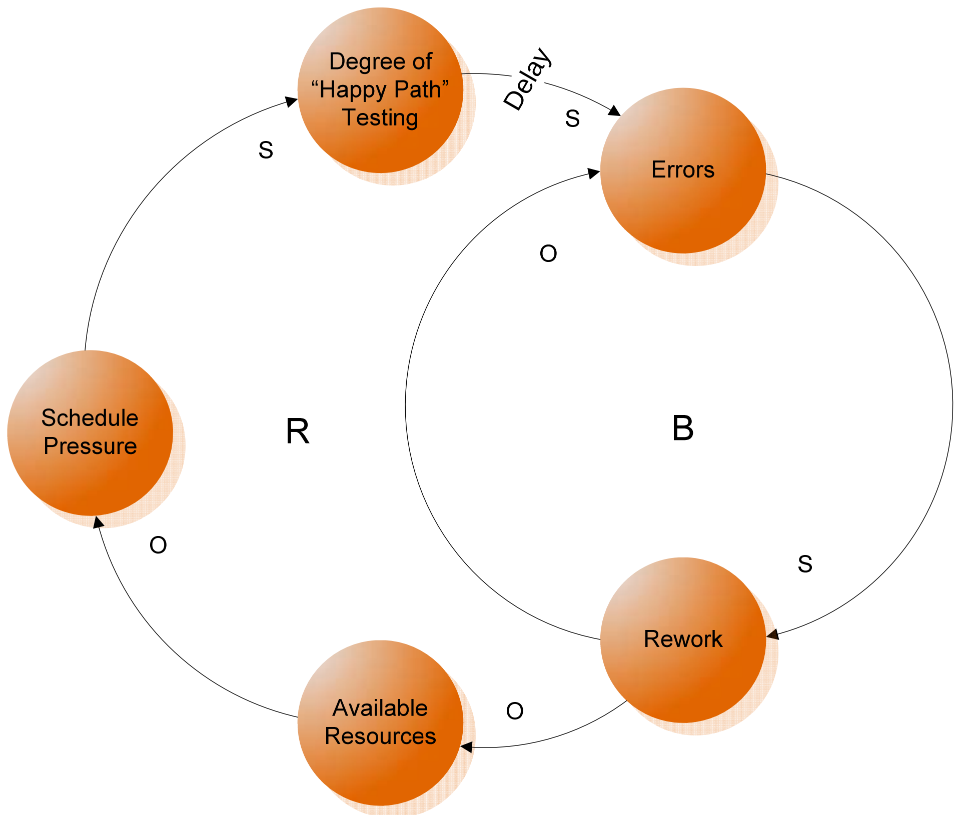
(Continued from page 1)

The Bigger Picture

Testing is a complex activity that, when done properly, employs many different tools and approaches. Testing must be planned and successfully executed on many levels (i.e., unit, integration, and system acceptance) in order to prove that the system is functioning properly before it is deployed. The purpose of scripted testing is to validate that the system is working as expected. Happy Path tests are typically tightly scripted tests of planned system functionality, and are a legitimate strategy for some aspects of testing—but not all. Complete and meaningful testing must also try to emulate the operational environment into which the system will be deployed. Comprehensive testing must attempt to break the system, generating errors in the way that normal users may do when they are using the live system, so that the consequences and probable system behavior can be understood.

Happy Path testing does *not* determine whether the system will behave well in the presence of errors; if used as the primary testing approach, the consequence is that undiscovered problems will still be present in the system. These errors will survive and multiply through successive development phases, and will ultimately be found either very late in development, or by users after deployment—when errors have the greatest impact, and are the most expensive to fix.

A Causal Loop Diagram of The Happy Path Testing Pattern



System variables (nodes) affect one another (shown by arrows): Same means variables move in the same direction; opposite means the variables move in opposite directions. Balancing loops converge on a stable value; Reinforcing loops are always increasing or always decreasing. Delay de-

Breaking The Pattern

No guide to software testing would advocate Happy Path testing *except* as a single element of a much larger and more comprehensive testing strategy. However, if the program testing budget is inadequate, or the available schedule for testing has been squeezed by prior schedule slips, it may become the only type of testing that can be completed within these constraints.

In trying to break out of the "Happy Path Testing" pattern, the program needs to first acknowledge that the fix they are using—testing only the system functionality that is expected to work—is only mitigating a *symptom* of the actual problem (i.e., abundant system defects).

Next, the program must commit to addressing the *fundamental* problem—finding the defects that will only occur when the system is actually used in the "real world," or when there are problems in the system's operating environment.

Several actions can help prevent Happy Path testing:

- Ensure that both resources and schedule are sufficient to provide comprehensive program testing coverage—and that they remain that way throughout the program.
- Require robustness testing that tests system behavior in the presence of input errors, bad data, and problems with the operational environment (network connectivity and similar factors).
- Test entire end-to-end operational scenarios, rather than only specific functions of the system. Problems are more likely to occur when multiple operations are performed in conjunction with one another, rather than in isolation.

[Abdel-Hamid 1991] Abdel-Hamid, T.K. & Madnick, S.E. *Software Project Dynamics: An Integrated Approach*. Prentice Hall, 1991.

Software Engineering Institute
Carnegie Mellon

AA11 091130

Acquisition Archetypes is an exploration of patterns of failure in software acquisition using systems thinking concepts. It is published by the Acquisition Support Program of the Software Engineering Institute.

For more information, visit http://www.sei.cmu.edu/acquisition/research/archetypes.cfm