Acquisition Archetypes

Changing Counterproductive Behaviors in Real Acquisitions

Firefighting

Good Intentions

When a project begins, no one *intends* to deliver it late, or to overrun their budget, or to give users a buggy system. It just seems to happen—all too often, and despite the best of intentions. Actually, though, the problems that envelop so many software acquisition efforts are predictable—which means that they are also avoidable, and often correctable.

We're going to explore one of those predictable patterns—one called *firefighting*. A recent government development program fell prey to it after mistakes were made in the earliest estimates of the work by the contractor.

Do You Smell Smoke?

In this project, mixing the contractor's poor estimation process with an aggressive schedule from the government yielded significant underestimation of the effort needed to develop each system release. Looming deadlines, and the probability of missing them, multiplied the schedule pressure, and work on the project became frenzied. A QA analyst observed that "the contractor burned hours like there's no tomorrow," yet productivity and quality fell off with increased overtime. The result: "They ended up rubber stamping code at code reviews."

> *"There are just too many unpredictable factors and variables to accurately estimate the effort required."*

When system acceptance testing finally started, the team found the current release had a high failure rate in test cases. The government technical lead admitted the project was behind schedule "because of all kinds of bugs."

Fire! All Hands on Deck!

The contractor's solution? Firefighting. Pull everyone off their assigned tasks to fix the problems blazing throughout the project. Resources were pulled off of every *other* effort that was going on in parallel—notably the next release.

Later, a team member noted that no task was safe from being stripped of people. The government acknowledged that delays on the current release would unquestionably affect the next release. The firefighting, he said, "sets my colleagues doing the next release up to fail, because then they have to start late, and their schedule will slip from the beginning."

> *"The contractor burned hours like there's no tomorrow"*

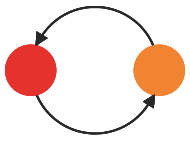
The contractor wanted to break out of this dynamic, but with all the people needed for estimating the next release busy fighting fires, "we're never able to get out ahead of the problems."

A Towering Inferno

So, the problem just got worse, and the flames hotter and higher. The contractor noted that the government deferred problem requirements—moving them to a new release—rather than facing the problem and cancelling or postponing indefinitely. The problem thus perpetuated itself, with deferred requirements mapped to future releases, and resources diverted from early release development.

The program manager, reviewing the smoking ruin of the development plan, summed it up.

"The first-order effects of what went wrong on the release were bad enough," he said. "It was late and over budget. But the contractor didn't want to acknowledge that that caused the *next* release to slip, and may have reduced functionality in the current release—leaving this [mess] on the side that someone has to clean up."



(Continued on page 2)

(Continued from page 1)

The Bigger Picture

There are many ways the firefighting dynamic can begin, but once started it is self-perpetuating. The initial trigger may be due to scope creep, budget cuts, underestimation of the actual effort, or other reasons. Processes are stressed, and short cuts may be taken in quality processes. This allows defects to survive or be inserted into the system.

Reading The Causal Loop Diagram

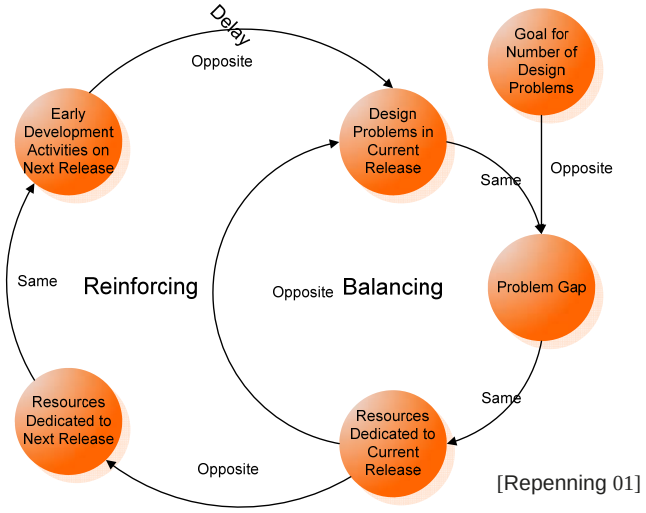
A program has a desired goal for the number of allowable defects in the delivered system—and the difference between that goal and the actual number of problems is the *Problem Gap* (see diagram). If this gap increases, then *Resources Dedicated to Current Release* must increase to do rework to fix problems.

More resources doing rework means fewer *Design Problems in Current Release*, and reduces the *Problem Gap*. This is a *Balancing* loop in which rework offsets (balances) the defects being inserted. Unless the staff size increases, more people assigned as *Resources Dedicated to Current Release* leaves fewer *Resources Dedicated to Next Release*. This reduces the resources available for *Early Development Activities on Next Release*—which, after a delay, increases the number of *Design Problems* in *Current Release*.

This exemplifies the classic problem of trading off long-term benefits for short-term gains, and results in exacerbating problems rather than resolving them. By "robbing Peter to pay Paul," additional resources will have to be spent in resolving the new problems introduced into the future releases.

"How can I break this vicious cycle of schedule slips, cost overruns, and high defect rates?"

A Causal Loop Diagram of the firefighting effect.



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* denotes actual time delays.

Breaking The Pattern

From a systems thinking perspective, to break out of this ongoing dynamic this program needs to: (1) acknowledge up front that the "fix" they are using—namely diverting resources to address problems in the current release—is just alleviating a *symptom* of the true problem, and (2) commit to solving the *real* problem—accurately estimating the time and effort required for a release, and staffing each new release in accordance with that estimate from the beginning so that more problems with quality don't occur [Kim 93].

For other programs that have not yet experienced this type of behavior, there are ways to avoid it [Repenning 01]:

- Don't invest in new tools and processes if you're already resource-constrained.
- Aggregation of resource planning (across all subtasks) is critical to fire prevention.
- When a project does experience trouble in the later phases of the development cycle, don't try to "catch up"—revisit the product plan instead.
- Don't reward developers for being good firefighters.

[Kim 93] Kim, Daniel H. *System Archetypes: Diagnosing Systemic Issues and Designing High-Leverage Interventions*, Vols. I, II, and III. Pegasus Communications, 1993.

[Repenning 01] Repenning, Nelson P.; Goncalves, Paulo; & Black, Laura J. *Past the Tipping Point: The Persistence of Firefighting in Product Development.* California Management Review, July 1, 2001.

Carnegie Mellon Software Engineering Institute

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/programs/acquisition-support/