Solving The Agile Catch 22

Prepared By: Critical Logic

WHITE PAPER

Critical Logic
PREVENT AND DETECT

_solving the agile catch-22

The Agile Revolution

When 17 software developers put pen to paper and released the "Manifesto for Agile Software Development" in 2001, they solidified a moniker for lightweight software project management and development methods that had been slowly growing in popularity since the 1970s. Since the Manifesto and the formation of the Agile Alliance, agile software methods have proven their real world value across numerous organizations and development projects worldwide.

Agile methods are credited with producing a variety of beneficial results including lower costs, higher quality, faster development, and increased revenue. All of these advantages are typically attributed to agile's user-oriented focus on requirements and its drastically shortened development cycles. Short sprints where software features are completed and tested enable continuous integration and, in theory, accelerate business value by constantly producing deployable software.

Agile's impressive advantages are typically attributed to its user-oriented focus on requirements and drastically shortened development cycles.

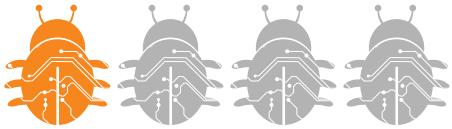
So Why Isn't Agile Perfect Yet?

These agile methods, however, are not beyond reproach. Countless projects have shown that tradeoffs and sacrifices are often made to garner the short development cycle benefits from an agile approach. For instance, the fundamentally lightweight requirements demanded by agile to keep up the pace of sprints very often are too light and result in ambiguity that shows up in test, or worse, in production. Moreover, cross-sprint and cross-team changes require a higher degree of coordination to keep test cases updated and avoid accidental breakages that are typically masked until integration testing.

In the best cases, teams find a mix of agile and other methods to try to account for organizational or project specific needs. But more often than not, issues that arise running agile projects become compromises or defects in the end product. In Agile, pace and complexity tend to overwhelm quality.

In response to these agile challenges, several options have been devised to plug the testing holes opened by ever-shortening development cycles. Risk-based testing attempts to mitigate critical or catastrophic failures by prioritizing feature and function tests based on their risk of failure. While this aligns well with the product backlog prioritization element of agile development, robust outcomes require high accuracy in the prioritization step, and the approach inherently accepts defect escapes by surrendering to incomplete test coverage.

Automated unit testing is another response to agile testing shortfalls, but data shows only 25-30% of software bugs are found with this method making it entirely inadequate _ by itself _ for critical software functions. Unit testing, with tools such as JUnit, simply moves the bulk of testing burden downstream to integration testing where the defect costs increase. Basically , automated unit testing is not a complete solution.



only 25-30% of software bugs are found with automated unit testing

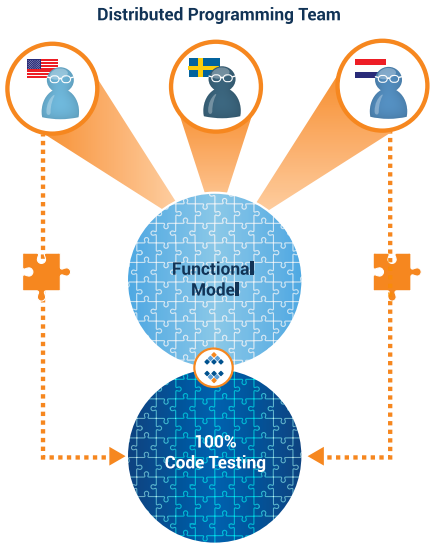
There have also been moves to perfect regression testing to close the testing gap. Quickly, however, maintenance of your regression test library becomes infeasible as the test cases grow with each agile evolution of the software. Cross-sprint or cross-team software changes exponentially increase your test cases, scripts and execution.

Continuous integration (CI) testing is desperately needed as test cases increase, but again, the pace of agile development cycles rarely permits the investment needed to achieve it. On complex systems, completely automated or "zero-touch" Continuous Integration is a proposed solution akin to "unobtainium" (elemental symbol $\sqrt{\pi}$). The concept of automated CI is nearly impossible to achieve when test cases and scripts are still manually updated in coordination with software code for both unit and integration tests, and the same cross-team or cross-sprint issues raise their heads again.

So, we appear to face a Catch-22 _ we must abandon agile methods that accelerate business results (through shorter development and release cycles) to achieve adequate requirements definition, testing coverage, and quality, or we must wrap our agile approach in testing methodologies that slow the process and eliminate the agile benefits. But appearances can be deceiving.

Could Modeling Be the Missing Link?

Enter cause-effect modeling (CEM). By making an adjustment to the starting point of each agile cycle - starting with a functional model - we can solve multiple problems simultaneously. First, we take a giant leap toward zero-touch integration testing where a model updated during an agile cycle automatically updates or generates test cases and scripts that can be executed automatically. By first creating a central model, we enable automatic updating of all regression tests in the library to eliminate the issues arising from cross-team and cross-sprint code changes and accidental breakages. When the models update, the tests update…period. And we maintain system-level awareness even when teams are siloed inside their development cycles.



Another obvious result of starting with a cause-effect model is the built-in elimination of ambiguity in requirements. Even at the pace of agile development and using lightweight user requirements, the coding requirements become clear as decisions are made to fill in gaps in the model. But with the model, the decisions are made *in situ* with the developers, and not in a massive upfront predictive process. Those requirements can still be repetitively validated through user review after short cycles.

By making an adjustment to the starting point of each agile cycle - starting with a functional model - we can solve multiple problems simultaneously.

One-hundred percent test coverage is no longer a drag on cycle speed when starting from a cause-effect model. When fast updates are made to the cause-effect model, the test cases are updated or created automatically, opening the door for true automated CI testing without the massive burden of manual test case scripting and maintenance. In the case of risk-based testing, 100% test *case* coverage means you know what you're electing not to test. So, catastrophic oversights in the prioritization process can be more easily avoided. In other words, you prioritize among 100% of the available test cases.

The CEM Payoff

The adoption of cause-effect modeling (CEM) is following a similar path to the agile software development philosophy and methods discussed here. The concepts of cause-effect modeling and functional engineering are not new but have been gaining popularity since the 1970s. In the 2000s, better tools and resources have become available to help software teams implement modeling in whatever development process they use: agile, XP, RAD, spiral, even waterfall. The ever-shortening development cycles in agile approaches, however, make agile a prime candidate for winning big using CEM.

One-hundred percent test coverage becomes possible while maintaining the fast pace of agile cycles

In review, if automated continuous integration testing is a major goal in your agile environment, cause-effect modeling enables the elusive automated test case updating, scripting automation, regression library updates and execution that leads to zero-touch CI. Lightweight user requirements no longer hinder complete features and test coverage as rapid model building fills in the ambiguities that lead to bugs. One-hundred percent test coverage becomes possible while maintaining the fast pace of agile cycles , and even risk-based testing effectiveness is drastically increased.

More Information

To learn more about real world success of CEM at work, see our case study of Sun Microsystems' (now Oracle) utilization of Critical Logic's CEM Editor to accelerate the continuous agile development of their product configurator software. The configurator manages more than 60,000 business rules and sees 25% of those rules change in every release which was done monthly before CEM was implemented. After adopting CEM and using the model as the starting point of every cycle, Sun reduced the release cycle from 4 to 2 weeks, effectively updated and executed 20,000 automated model-driven tests per cycle, experienced a 95% reduction in defect escapes, and did all of it with 30% fewer testing resources.

Contact Critical Logic to find out how your teams can take advantage of CEM in your agile development environment. Call Critical Logic at 415.814.9524 or visit us at www.Critical-Logic.com.