7

September, 2009

testing experience

The Magazine for Professional Testers

printed in Germany
print version 8,00 €
free digital version
www.testingexperience.com
ISSN 1866-5705

Agile Testing

How Agile Methodologies Challenge Testing

by Rex Black

This article is excerpted from Chapter 12 of Rex Black's upcoming book Managing the Testing Process, 3e.

A number of our clients have adopted Scrum and other Agile methodologies. Every software development lifecycle model, from sequential to spiral to incremental to Agile, has testing implications. Some of these implications ease the testing process. We don't need to worry about these implications here.

Some of these testing implications challenge testing. In this case study, I discuss those challenges so that our client can understand the issues created by the Scrum methodology, and distinguish those from other types of testing issues that our client faces.

In my books and consulting, I typically recommend a blended testing strategy, consisting of three types of test strategies:

- Analytical risk-based testing;
- Automated regression testing;
- Reactive testing (also referred to as *dynamic testing*).

The blended testing strategy I recommend aligns well with Scrum and other Agile methodologies. In some cases, this strategy will mitigate the testing risks and reduce the testing challenges associated with these methodologies. However, it does not resolve all of the risks and challenges. In this article, let's examine some of the challenges that I've observed with RBCS clients using Scrum and Agile methodologies.

Dealing with the Volume and Speed of Change

One of the principles of Agile development is that project teams should "welcome changing requirements, even late in development" (see agilemanifesto.org). Many testing strategies, especially analytical requirements-based testing, become quite inefficient in such situations.

However, risk-based testing accommodates change, since we can always add risks, remove risks, change risks, and adjust the level of risk. If test execution is underway, we can adjust our plan for the remaining period based on this new view of quality risk. Since risk-based testing provides an intelligent way to decide what to test, how much, and in what order, we can always revise those decisions based on new information or direction from the project team.

Smart automated testing also accommodates on-going change. With care, automation at the graphical user interface can be maintainable. Automation at stable command-line interfaces does not tend to suffer severe maintainability problems.

The reactive testing that I recommend, not requiring much documentation, is also quite resilient in the face of change.

However, change can impose challenges for testing that are independent of the use of these test strategies. Many of these challenges arise from change in the definition of the product and its correct behavior (see also below). When the test team is not kept informed of these changes, or when the rate of change is very high, this can impose inefficiencies on the development, execution, and maintenance of tests.

Remaining Effective during Very Short Iterations

In sequential lifecycles, test teams can have a long period of time in which to develop and maintain their tests, in parallel with the development of the system, prior to the start of system test execution. Some more formal iterative lifecycle models, such as Rapid Application Development and the Rational Unified Process, often allow substantial periods of time between test execution periods for each iteration. These intervals allow test teams develop and maintain their test systems.

Agile methodologies like Scrum are less formal and faster moving. Consistent with the evocative name, *sprints*, these methodologies use short, fast-paced iterations. For a number of clients, RBCS consultants have seen this pace and brevity further squeeze the test team's ability to develop and maintain test systems, compounding the effects of change noted earlier. Testing strategies that include an automation element have proven particularly sensitive to this challenge.

The risk-based element of the recommended strategy can help. Risk-based testing focuses on the important areas of test coverage, and de-emphasizes or even cuts less important areas, relieving some of the pressure created by the short iterations. This ability to focus proves especially helpful for test teams also under tight resources constraints. Test teams in an Agile world should develop, maintain, and execute tests in risk priority order. Using risk priority to sequence development and maintenance efforts allows the test team to have the most important tests ready at the beginning of each sprint's test execution period.

Receiving Code after Inconsistent and Often Inadequate Unit Testing

Many of the authors, practitioners, and academics involved with Agile methodologies stress good, automated unit testing. Open-source test harness such as J-Unit and Cpp-Unit minimize the tool costs of doing so, overcoming one key obstacle to automation. Such automated unit

tests allow what Agile proponents call *refactoring*. Refactoring is the redesign of major chunks of code or even entire objects. The automated unit tests provide for quick regression testing of refactored code. Some Agilists recommend substituting automated unit tests for design, as in Test Driven Development. While this sounds good in theory, there are two problems that we tend to observe with this approach in practice.

First, unit testing has limited bug-finding utility. Capers Jones has found 25 to 30% average defect removal effectiveness for unit testing.[1] RBCS assessments have shown that good system testing by an independent test team averages around 85% defect detection effectiveness. So, while unit testing helps, the main filter to prevent excessive field failures remains system testing.

Second, we find that, under the guise of excuses both valid and not-so-valid, many programmers do not create automated unit tests and in some cases don't do any unit testing at all. This creates a great deal of trouble for the system test team, which, as mentioned above, remains a critical bug-removing filter. The short test execution periods on Agile sprints, compared to sequential projects, means that the degree of damage caused by one or two days' of test progress blockage due to highly buggy code is higher than in a sequential project.

Delivery of unstable, buggy code will undermine one of the key benefits of the risk-based testing portion of the recommended strategy, which is the discovery of the most important defects early in the test execution period. It also inevitably leads to a large degree of code churn during testing, since so much must change to fix the bugs. The amount of change can ultimately outstrip even the ability of the best automated regression test system to keep up, which would then lead to lower defect detection effectiveness for the test team.

Managing the Increased Regression Risk

Capers Jones has found that regression accounts for about 7% of bugs.[2] In iterative lifecycles like Scrum, though, code that worked in previous sprints gets churned by new features in each subsequent sprint. This increases the risk of regression. Agile methodology advocates emphasize good automated unit testing in part to manage the regression risk inherent in such churn.

However, good unit testing has limited defect removal effectiveness, as cited earlier. So, automated regression testing via unit tests will likely miss most of the regression bugs. Therefore, we need effective regression testing at the system test level (which has a higher level of defect detection effectiveness). By combining risk-based testing with the automated regression testing, test teams can effectively manage the increased regression risk.

Making Do with Poor, Changing, and Missing Test Oracles

Agile methodologies de-value written documentation. Special scorn is reserved for specifications. For example, the Agile Manifesto suggests people should value "working software over comprehensive documentation." This creates real challenges for a test team. Testers use requirements specifications and other documents as *test oracles*; i.e., as the means to determine correct behavior under a given test condition. We have seen testers in Agile situations given documents with insufficient detail, or, in some cases, given no such documents at all.

Even the project team provides the test team with adequate documents, two other Agile development principles keep the test oracle challenge alive. First, Agile requires teams to embrace change, as I discussed earlier. Second, Agile principles state that "the most efficient and effective method of conveying information to and within a development team is face-to-face conversation" (see agilemanifesto.org). For many of our clients following Agile methodologies like Scrum, these two principles allow the project team to change the definition of correct behavior at any time, including after testers have tested to confirm a particular behavior and after testers have reported bugs against a particular behavior. Further, the definition of correct behavior can change in a meeting or a discussion which might not involve the test team and which might produce no documented record of the change.

No known test strategy can resolve this challenge. Resolving the challenge requires change management. The percentage of rejected bug reports provides a measurable symptom of this challenge. Rejected bug reports are ones that the team ultimately discarded because the reports described correct behavior. Projects with well-defined, stable test oracles enjoy bug reject rates below five percent. Projects with poor, changing, or missing test oracles often endure bug reject rates of 30 percent or more.

We have estimated imposed test team inefficiencies from such test oracle problems at around 20 to 30 percent. Further, the inability to determine precisely whether a test failed affects both the efficiency of the testing and the defect detection effectiveness. When testers spend time isolating situations that the project team ultimately chooses to define as correct behavior, that takes away time they could have spent finding and reporting real bugs. These bugs create subsequent problems customers, users, and technical support staff, and distractions for developers and test teams.

Further, the situation creates frustration for the testers that reduces their morale and, consequently, their effectiveness. Testers want to produce valid information. When much of the information they produce – in the form of rejected bugs reports – ends up in the figurative wastebasket of the project, that tends to make people wonder why they bother.

It's important to realize that this reduction in test effectiveness, efficiency, and morale is a potential side-effect of Agile methodologies. Organizations using Agile methodologies must assign responsibility for these outcomes in the proper place. Bad problems can get much worse when the test team is held accountable for outcomes beyond their control.

Dealing with a Shifting Test Basis

Requirements-based testing strategies cannot handle vague or missing requirements specifications. Missing requirements specifications would mean a test team following a requirements-based testing strategy not only can't say what it means for a particular test to pass or fail, they wouldn't have a *test basis*. The test basis is that which the tests are based upon. Requirements-based testing strategies require test teams to develop test cases by first analyzing test conditions in the requirements, and then using those test conditions to design and implement test cases. A missing or poor requirements specification won't work for such analysis.

The test basis also provides a means to measure the results. In a requirements-based test strategy, testers can't report test results accurately if the requirements are missing or poor. Testers can't report the percentage of the test basis covered by passed tests, because the requirements won't provide enough detail for meaningful coverage analysis.

With the risk-based testing strategy recommended in my book, test teams can evade both problems. For the test basis, testers use the quality risk items. They design and implement test cases based on the quality risk items. The level of risk associated with each risk item determines the number of test cases and the priority of the test cases derived from that risk item. The test team can report test results in terms of quality risks mitigated versus not mitigated.

From Detailed Documentation to Many Meetings

As the Agile Manifesto quotation cited earlier puts it, people should focus on creating working software rather than comprehensive documentation. However, the information that was, under the *ancien regime*, captured and exchanged in these documents must flow somehow, so Agile advocates promote "face-to-face conversations." This, of course, is another name for a meeting. From a marketing perspective, it was smart of the Agile advocates not to use the word *meeting* in the Agile Manifesto, but the reality remains.

I mentioned in an earlier section on test oracle issues with Agile methodologies the problem of a meeting or a discussion that changes the definition of correct behavior, which did not involve the test team, and which did not produce a documented record of the change. The flip side of this problem, in some organizations, is that everyone gets invited to every

1 See Jones' article "Measuring Defect Potentials and Defect Removal Efficiency," found in the Basic Library at www.rbcs-us.com.

2 See, for example, Jones' figures in Estimating Software Costs, 2e.

meeting, the meetings balloon in number and duration, managers and leads are not available to manage and lead their team because they are in meetings much of the day, and effectiveness and efficiency drop.

One manager jokingly described this situation as follows: "Scrum is a heavyweight process. I'm surprised at the name *Agile* – it should be called *couch potato*. There are too many meetings. There's too much jawboning. I find it really ironic that there are all these books explaining how simple it is."

To be fair, having too many meetings is a problem that any project following any lifecycle model can suffer from. I've worked on classic waterfall projects as a test manager where I spent four hours or more per day in meetings. I had one client relate a hilarious anecdote where a senior manager, in response to a complaint from a line manager about how attending meetings was negatively impacting his ability to lead his team, shouted, "We're going to continue to have these meetings until I find out why nothing is getting done around here!"

That said, every organization, every project, and every lifecycle has to strike the right balance. In some cases, organizations and projects following Agile methodologies react too strongly to the Agile Manifesto's comments on documentation and face-to-face "discussions." Further, embracing change should not meet throwing out or re-considering previous decisions to the extent that it paralyzes the team. Teams using Agile methodologies must achieve the right balance between documentation and meetings. Teams using Agile methodologies must have crisp, efficient meetings that move the project forward and course-correct, not meetings that grind the team down and go around in circles.

Holding to Arbitrary Sprint Durations

Some of our clients following Agile methodologies like Scrum tend to ritualize some of the rules of the process. The time deadlines for sprints seem particularly subject to this ritualization. At first, I was puzzled that these same clients discarded some other rules, such as the requirement for unit testing, often with less reason than the reasons for which they held tightly to the deadlines. However, deadlines are tangible, while the benefits of unit tests are not as well-understood and clear, so I can now see the reasons behind the selective emphasis on certain Agile process rules over others.

For example, suppose that a project team follows four week sprints. A systemic problem in our industry relates to software estimation, particularly the tendency to over-commit in terms of the number of features (*user stories*) for that sprint. So, on the last Friday of the sprint, with development ending late for the sprint, the arbitrary deadline remains intact at the expense of the test team's weekend.

Fully resolving this challenge requires team and management maturity. When people habitually and systematically over-commit, management should require use of historical progress metrics for estimation. Many Agile practitioners use metrics like *burndown charts* and *story-point velocity*. The Scrum process includes gathering and using such metrics.

If fixing the software estimation problems cannot occur, risk-based testing helps the test team deal with systematic and constant over-commitment. To start with, when the test team is time-crunched over and over again at sprint's end, the test team should accept that the project team's priorities are schedule-driven, not quality-driven. The test team should revise the risk analysis approach to institute an across-the-board reduction in the extent of testing assigned to each quality risk items during risk analysis for subsequent projects. That way, at least the test team won't over-commit.

In some cases, in spite of reducing the scope of testing, the test team still can't execute all the tests in the available time at the end of a sprint. If so, rather than ruining their weekend to run every test, the test team can select the most important tests using the risk priority number. Less important tests can slip into the next sprint. (You'll notice that Scrum and other Agile methodologies allow user stories to slip from one sprint to the next in the same way.) Of course, if the test team must consistently triage its tests in this fashion, they should again adjust the test extent mapping downward for future sprints.

Dealing with Blind Spots in the Sprint Silos

Not all experts on Agile methodologies agree on the need for independent test teams. Some seem to think of testing as a subset of the tasks carried out by programmers on an Agile team, specifically creating various forms of automated unit tests and/or acceptance tests. This might exclude an independent test team, or it might transform the role of that team.[3]

Most of our clients adopting Agile methodologies have retained the independent test team, or at least the independent tester, to some extent. For those retaining the independent team, most have chosen to partition the team across the sprints in some way, often making each tester answerable on a day-to-day task basis to the Scrummaster or other sprint leader, rather than to the test manager. In the case of having independent testers but not independent test team, there is no test manager.

This provides some advantages, especially to the person leading the sprint:

- Each tester focuses entirely on sprint-related tasks, with minimal outside distractions.
- Each tester allocates time entirely to the benefit of the sprint and its exigent goals.
- The sprint leader can re-direct any tester to focus on what is most important to the sprint team, often without having to consult the test manager.
- The sprint leader can – and, at the end of a sprint, often will – call on the tester to put in extra efforts to hit sprint targets.
- The amount of test effort allocated to the sprint does not vary once the number of testers for the sprint is determined, and the test manager cannot surprise the sprint leader with a sudden reduction in test effort to serve other test team priorities.

As you can see, some of the advantages have zero-sum-game elements that carry within them the seeds – if not the actual fruit – of various potential problems.

The challenges of this approach – some of which are obvious corollaries of the advantages – include the following:

- The tester thinks of himself as – and conducts himself as – less of a tester and more of a specialized developer. This can include a loss of interest in growing test-specific skills in favor of growing technical skills.
- The tester has less contact with people outside the sprint team, reducing the wider, system-level perspective provided by an independent test team.
- In the absence of coordinative guidance from the test manager, the tester starts to make mistakes related to gaps and overlaps. The tester fails to perform some test tasks that make sense, especially longer-term investments that don't necessarily benefit the current sprint. The tester redundantly performs test tasks done by other testers on other sprints, because she wasn't aware of the other testers' work.
- The test manager, having lost the ability to manage the workload of their resources, finds the morale suffers and turnover increases as unsustainable workloads at the end of each sprint take their tool.
- The ability of the test team to grow a consistent, powerful, maintainable test system – the test scripts, the test tools, the test data, and other stuff needed for effective and efficient testing in the long-term – goes down because of the exigent focus on the sprint's immediate needs.
- Testers display a tendency to "go native" and lose some of the objectivity that an independent test team normally provides, suffering a loss of defect detection effectiveness.

None of these challenges (or advantages, for that matter) arises from Agile methodologies

3 See, for example, the abstract of Kent Beck's talk, given at the Quality Week 2001, conference, www.soft.com/QualWeek/QW2001/papers/2Q.html. Beck claimed that unit testing by programmers might eventually make the defect detection role of independent test teams (referred to as "QA" in the abstract) superfluous, transforming testing into a role similar to a business analyst's. My earlier comments about unit testing, and the figures I cited from Caper Jones' work on the limits of the effectiveness of unit testing, make me skeptical that we will see substantially bug-free code delivered to test teams, no matter what the lifecycle model, but I'd enjoy working on projects where someone managed to prove me wrong.

per se; it just happens that Agile methodologies as typically practiced tend to accentuate them. I have observed similar problems for years on projects following sequential models or no model at all, when the test team adopted what I refer to as the *project resource* approach to test team organization.

The challenges are not insurmountable, if an independent test team exists. The test team, lead by a good test manager, can introduce centripetal forces that will bind the team together and makes its actions consistent and congruent. These forces then balance the sprint-specific centrifugal forces that tend to push the test team members into sub-optimizing for their sprint as well as isolating the test team members from broader testing considerations.[4]

One approach to managing these challenges is to have a separate test period that follows the development sprint. (To keep the terminology clean, some like to call this approach an alternation of *development sprints* following by *test sprints*.) Some of our clients have pursued this approach. It does seem to work for them, when the test team remains engaged in the development sprint. In other words, if the test team disengages from the development team during the development sprint, you are just exchanging one form of siloing for another.

Managing Expectations

To close this article, let's look at a psychological challenge to test teams on Agile projects. Gartner, the industry analysts, say that IT industry adoption of new technologies and ideas goes through a *Hype Cycle* (see www.gartner.com/pages/story.php.id.8795.s.8.jsp). The Hype Cycle consists of five distinct phases, as they describe on their Web site:

1. **Technology Trigger**. The first phase of a Hype Cycle is the "technology trigger" or breakthrough, product launch or other event that generates significant press and interest.
2. **Peak of Inflated Expectations**. In the next phase, a frenzy of publicity typically generates over-enthusiasm and unrealistic expectations. There may be some successful applications of a technology, but there are typically more failures.
3. **Trough of Disillusionment**. Technologies enter the "trough of disillusionment" because they fail to meet expectations and quickly become unfashionable. Consequently, the press usually abandons the topic and the technology.
4. **Slope of Enlightenment**. Although the press may have stopped covering the technology, some businesses continue through the "slope of enlightenment" and experiment to understand the benefits and practical application of the technology.
5. **Plateau of Productivity**. A technology reaches the "plateau of productivity" as the benefits of it become widely demonstrated and accepted. The technology becomes increasingly stable and evolves in second and third generations. The final height of the plateau varies according to whether the technology is broadly applicable or benefits only a niche market.

As we approach the 2010s decade, we are seeing Agile methodologies in the peak of inflated expectations phase. RBCS clients have inflated expectations for Agile methods that relate to quality, productivity, and flexibility.

Some test teams on Scrum projects report to management that the product quality is no higher than normal, and sometimes even lower than normal. Some test teams on Scrum projects report to management that the challenges discussed in this appendix have reduced their efficiency. Some test teams on Scrum projects report to management that, while development might not experience negative consequences or pain from change – which is a key promise of Agile methodologies and one of the key management selling points – testing will still endure problems when coping with unlimited, unmanaged change.

When these test teams on Scrum projects report these negative outcomes to management, management experiences a psychological phenomenon called *cognitive dissonance*. Cognitive dissonance involves feelings of mental tension between the incompatibility of their expectations of Agile methodologies and the observed results. Ultimately, these cognitive dissonance experiences, across the various adopters of Agile methodologies, will push these approaches along the Hype Cycle, out of the peak of inflated expectations. In the short run, though, management might engage in another psychological phenomenon called *projection*. This involves projecting onto others how you feel about something they are associated with, but perhaps not responsible for. The experienced test professional knows this phenomenon better under the phrase *killing the messenger*.

Conclusion

The test strategies I typically recommend will support the stated goals of Agile methodologies. Risk-based testing supports increased quality, since it focuses testing on high-risk areas where testing can significantly reduce the risk. Risk-based testing supports increased productivity, since it reduces or eliminates testing where the quality risk is lower. Risk-based testing supports flexibility, since it allows regular revision of the quality risk items which re-aligns the remaining testing with the new risks and their new levels of risk. Automated regression testing helps to contain the regression risks associated with Agile methodologies, allowing a higher rate of change. Reactive testing allows testers to explore various aspects of the system that risk-based testing and automated regression testing together might miss.

However, this blended risk-based, automated, and reactive test strategy cannot fully resolve the challenges covered in this case study. In the long run, people will come to recognize that, and rational trade-offs will prevail. In the short run, though, while Agile methodologies remain on the peak of inflated expectations, the test team must be careful to communicate any testing issues that arise due to Agile methodologies and their effect on testing rather than from testing itself.

4 See Chapter 8 of my book Managing the Testing Process. The first edition (1999), the second edition (2003), and the upcoming third edition (2009) all make the same point. Plus ca change, plus c'est la même chose...

Biography

With a quarter-century of software and systems engineering experience, Rex Black is President of RBCS (www.rbcs-us.com), a leader in software, hardware, and systems testing. For over a dozen years, RBCS has delivered services in consulting, outsourcing and training for software and hardware testing. Employing the industry's most experienced and recognized consultants, RBCS conducts product testing, builds and improves testing groups and hires testing staff for hundreds of clients worldwide. Ranging from Fortune 20 companies to start-ups, RBCS clients save time and money through improved product development, decreased tech support calls, improved corporate reputation and more. As the leader of RBCS, Rex is the most prolific author practicing in the field of software testing today. His popular first book, Managing the Testing Process, has sold over 35,000 copies around the world. His five other books on testing have also sold tens of thousands of copies. He has written over twenty-five articles, presented hundreds of papers, workshops, and seminars, and given about thirty keynote speeches at conferences and events around the world. Rex is the President of the International Software Testing Qualifications Board and a Director of the American Software Testing Qualifications Board.