david consulting group
TRUSTED ADVISOR

How can you make integration/acceptance testing truly Agile?

August 2013

Scope of this Report

The flow of testing is different in an Agile project. In many cases, organizations have either not recognized the change in flow, or have created Agile/waterfall hybrids with test groups holding onto waterfall patterns. While some of the hybrids are driven by mandated contractual relationships, the majority are driven by lack of understanding or fear of how testing should flow in Agile projects which leads to the mistaken belief that integration and acceptance testing can't be performed within Agile frameworks.

This report is intended to show where agile project integration and acceptance testing fit in the overall flow of Agile and why they are integral to effective Agile.

The Flow of Agile Testing

Testing is a major component of all Agile projects, but the flow of that work is different than in waterfall.

A typical waterfall testing flow, circa 1990s, was for a Quality Assurance (QA) manager to create an initial test plan during the project planning phase. Next, the business analyst, or someone acting in that capacity, would go off to gather the requirements. In most organizations no one from QA would participate during requirements elicitation. Instead, work was compartmentalized like an assembly line. When the requirements were complete, QA personnel would begin to build test scenarios, test cases and detailed test plans as developers built the software in parallel. In enlightened organizations, developers and QA might have interacted during reviews to approve documents or code reviews. When the developers were done building, the QA personnel would test the projects deliverables (often in whatever time the developers' overruns had left them) and point out the defects.

The flow of testing in a typical Agile project is much more integrated. The idea of a high-level test plan is generally subsumed into the Agile concept of the definition of "done". The definition of done includes the macro steps that will be required for each piece of work to be completed and ready to be implemented. A general definition of done for a unit of work might include definition, coding, unit testing, system testing, user acceptance testing, documenting (as provides value) and perhaps other macro categories of work. How each of these macro categories should be accomplished for a specific unit of work is defined contextually as a piece of work is accepted into a sprint during sprint planning.

The first two major test-planning events from the waterfall world, high-level test planning and writing test cases, are spread across the Agile project. As the Agile team breaks down the units of work into tasks and activities, the testing activities are defined and captured by the team. Many Agile techniques begin building software by detailing the test cases (all types of testing) for each unit of work. This happens as the team begins to understand the unit of work but before a single line of code is written. Conceptually, satisfying the tests cases proves the unit of work is "done" both at the code level

and function level. Another typical difference in the flow of testing in an Agile project is the amount of automated testing which occurs every time the software is built (in its simplest terms, a "build" is the process of putting the software together as a whole, as if it were to be put into production).

There are a myriad of hybrids of waterfall and Agile. For example, when a separate team is created for testing and that team picks up after the development team writes and unit tests the code. This hybrid holds on to the idea of an independent test team that runs tests and points out defects and shortcomings. It also creates or continues a separation between QA personnel and the sprint team which makes communication difficult. This scenario also increases the cost of rework because defects are found out-of-phase which means reopening units of work that were considered complete (and may have been used to build other units of work).This scenario almost never makes sense. A second, and perhaps more palatable hybrid, is the use of a "hardening" sprint in which the team performs a final set of tests. These tests sometimes include a final integration test or a final user acceptance test. This scenario is problematic as it requires building up a significant amount of work before review, thereby risking significant rework.

Hybrids are useful if testing has to be isolated for contractual reasons but they almost always generate rework and should be avoided.

The flow of testing in an Agile organization is different. The reason Agile embeds testing into the basic flow of work is to more effectively generate feedback for the entire team early in the process. Breaking the testing into parts that can be integrated into the day-to-day flow of work helps facilitate communication in an environment where testing is not someone else's responsibility. The flow of testing in Agile is an explicit recognition that good code is tested code.

User Acceptance Testing In Agile

User Acceptance Testing (UAT) in an Agile project is generally more rigorous and timely than the classic "end of project" UAT found in waterfall projects. The classic definition of a UAT is a process that confirms that the output of a project meets the business needs and requirements. As we have noted, in waterfall projects the UAT is usually the last step in the development process witht the resulting problem that significant defects are found late in the process or, worse, the business discovers that what is being delivered isn't exactly what they wanted. Agile projects provide a number of opportunities to interject UAT activities throughout the process, starting with the development of user stories, to the sprint reviews and demos and, finally, the UAT sprints at the end of a release. Each UAT opportunity provides a platform for active learning and feedback from the business.

Agile User Acceptance Testing begins when user stories are defined. A user story should include both story and acceptance test cases (also known as acceptance criteria). As noted in "Daily Process Thoughts", July 31, 2013, one technique for creating and grooming user stories is called "The Three Amigos." This technique gathers one representative from each of the business, professional testing and development (the 3 Amigos) so that all major constituencies are represented. As a user story is defined, so are the acceptance criteria. Adding the focus on business acceptance criteria during the definition of user stories begins the UAT process, rather than waiting until later in the project. Laying out the acceptance criteria when you begin work on a story creates a focus that helps the team to stay focused

on what is actually needed and reduces the potential for rework and gold plating (adding extra features).

A second layer of UAT activity in Agile projects is found in the "end of sprint" demonstration. The "Daily Process Thoughts", June 20, 2013, provided a description of typical demonstration (also known as a show 'n' tell or sprint review activities). As a reminder, demonstrations are planned so that the product owner and team can provide proof that they have solved the business need. Demonstrations are interactive so that stakeholders can provide feedback and buy into the solution or ask for something different. Another technique that many Agile teams add to their working process is adding UAT tasks for each story. This ensures another level of user interaction as part of the development process, thereby increasing feedback and acceptance. In both cases, the interaction at the end of the sprint is earlier than is common in waterfall projects.

The third level of UAT is the inclusion of a dedicated sprint to perform a final, overall UAT prior to release (this should be part of the release plan). In this scenario, the units of work defined for the sprint would be focused on test cases or scenarios and then fixing any discovered defects. Having a sprint focused on UAT has risks. The risk is that delaying UAT until a specific sprint increases the probability of finding problems that require substantial rework later in the cycle, where they are more costly to fix. This is the same basic issue with the placement of the UAT in a waterfall project. So why do it? One of the reasons this approach might be needed is where users need to validate the entire system before acceptance (this happens in life critical systems and in many regulated markets). Developing stories with acceptance test cases, ensuring interaction with stakeholders at demos and adding user acceptance tests task in order for stories to be completed will help minimize the need for a specific UAT sprint and mitigate most of the risks of this technique when you have to use it.

User Acceptance Testing is at least as rigorous in Agile project as in most waterfall development techniques. As importantly, Agile UAT techniques are applied much earlier in the development process providing earlier feedback to the team. Earlier feedback reduces rework by finding and fixing problems before they can get bigger. Many Agile UAT activities are part of standard Agile practices including acceptance test cases in user stories and generating interactions with stakeholders in demonstrations. When needed, extra activities are easily integrated into Agile development techniques such as adding UAT tasks in developing stories or include UAT specific sprints. UAT activities build trust with stakeholders. UAT also proves to stakeholders that were not involved in the Agile development process that project is delivering on the business needs. User Acceptance Testing is a necessary step in any project, Agile just spreads it out and does it earlier (and better).

Integration Testing In Agile

Integration testing is a type of testing in which components (software and hardware) are combined to confirm that they interact according to expectations and requirements. Integration testing is an important testing technique in any project and perhaps even more so in Agile projects and programs because it is core to the concept of the "definition of done." While important to Agile, actually performing integration testing can be difficult unless you have layered the technique into the Agile framework you use. To perform an integration test of any sort you need:

- a test environment that closely mirrors the target (or production) environment,
- a comprehensive build of the software,

- data that exercises the connections inside and outside the application and
- a test plan (including test cases).

These are the basic requirements for integration testing but there are all sorts of "nice to haves" that you can add such as test automation software, automated test scripts and an automated data build. You can survive without these extras but any test will take longer which means you will have lower velocity.

Without an environment to run an integration test, the results will at best be ambiguous. The test environment should mirror the hardware and software configuration that the project will run when delivered as closely as possible to ensure that the results you see are predictive. The goal is to understand how the software will work when the project is installed in production. Examples of differences between test environments and production include debugging and test monitoring tools. As many have learned the hard way (whether Agile or not), every substantive difference between production and test is a potential failure point when we implement.

The second requirement for adequate integration testing is a comprehensive build of the software. This is where Agile projects and programs with daily builds make integration testing much easier. "Continuous builds" (also known as "continuous integration") are an important Agile technique that ensures that the parts fit together as work is done. Continuous builds are nearly always coupled with either regression testing (which checks if the changes made have broken anything) and/or "smoke tests" (which check the major functions to make sure they still work).

The third requirement is data for testing. Having carefully crafted test data ensures that you actually are testing what you want and need to test. The data for an integration test should be generated prior to each test run. As projects add components or functions, the process that is used to create the test data must be updated or will risk missing testing the integration of a new component. The process of creating test data should be as rigorous as the creation of functionality that will be installed. One mechanism for accomplishing this level of rigor is to rebuild the test script and data as part of user story creation. The tests will fail until the new functionality is implemented.

The final, and perhaps most controversial, requirement is a test plan. A test plan of some sort is needed to make sure the team knows how they are going to ensure that all of the components (software and hardware) actually work together and then prove it. How the test plan is documented can be left to the discretion of the reader but it should include the approach, how the tests will be accomplished and how the test plan will be maintained. Note these can all be documented as tasks on a Scrum or Kanban board as easily as a Word document. We have seen automated integration test scripts that would suffice nicely for a test plan. The test plan should only include what is absolutely needed so the team can be assured that all of the application's components have been successfully integrated and that the application then integrates with the overall environment

Integration testing is an important part of how an Agile team works. In many organizations the teams spend their spare time building the test harnesses needed to automate the process so that it can execute outside of the teams' core work hours. Testing makes their life easier in the long run and helps them deliver more value. When organizations start to ask questions about how integration testing can work in an Agile environment, it is usually a sign. A sign that it is time to add some of the more technical techniques to the project management oriented Agile framework. The combination of continuous

builds with nightly regression and integration testing are hallmarks of a highly efficient Agile team. Can integration testing be done on an Agile project or program? The question that you might ask is, "How can you be Agile without thorough integration testing?"

Conclusion

One of the principles of the Agile Manifesto calls for delivering working software frequently. "Delivering frequently" infers that to deliver anything of significant size that the functionality delivered will need to be a group of small bits of functionality assembled together. In order for working software to be delivered frequently, the delivery needs to work and can't "break" what has been delivered previously. Breaking previously delivered software means rework. Every minute you spending doing rework or fixing defects is time that is not available to build new functionality. Avoidance or reduction of rework is why Agile and lean testing frameworks continue to build across the successive sprints.

The tests are built to ensure the functionality developed in each sprint works (or doesn't work), and that we have not broken anything as a consequence of our work. Tests generated from sprint to sprint build incrementally so that by the conclusion of the project each new function has had functional tests, integration tests and regression tests performed in overlapping waves. The continuous retests as we incrementally build the software generate highly rigorous testing overall.

We build and test incrementally in Agile to avoid nasty surprises. The feedback from testing keeps the development process on track.

Sources

1. Daily Process Thoughts, http://tcagley.wordpress.com/
2. Agile Manifesto, http://agilemanifesto.org/