Enabling Agile Testing Through Continuous Integration

Sean Stolberg
Pacific Northwest National Laboratory
Sean.stolberg@pnl.gov

Abstract

A Continuous Integration system is often considered one of the key elements involved in supporting an agile software development and testing environment. As a traditional software tester transitioning to an agile development environment it became clear to me that I would need to put this essential infrastructure in place and promote improved development practices in order to make the transition to agile testing possible. This experience report discusses a continuous integration implementation I led last year. The initial motivations for implementing continuous integration are discussed and a pre and post-assessment using Martin Fowler's "Practices of Continuous Integration" is provided along with the technical specifics of the implementation. The report concludes with a retrospective of my experiences implementing and promoting continuous integration within the context of agile testing.

1. Introduction

"Hi. My name is Sean and I'm software tester. I've been waterfall-testing-free for over a year now." *(Applause is heard, hugging observed).*

Ok, hopefully you found at least a little bit of humor in that first paragraph, but I really do feel like I've been in "Waterfall Testing Rehab" for over a year now. This experience report describes significant aspects of my journey transitioning from a more traditional "software QA" role to a more effective "agile software tester". Specifically, I will share my experiences implementing a Continuous Integration system to enable the transition to agile testing techniques and approaches.

As a software tester that had worked in a traditional waterfall software development environment for eight years, I took my first job with an agile development team in November of 2007. I had no idea the changes that lay ahead of me in testing.

For my first two sprints I tried applying traditional testing approaches (test plan, clarify requirements, write test cases, test case review, etc) but kept coming up very short on time, and thus coverage, by the end of the iteration. "Technical testing debt" was accumulating in the form of manual regression tests needing to be run at the end of every sprint. We just didn't have time to run them. In summary, all of the instinctive ways I knew how to test were not holding up in the context of short iterations delivering new functionality.

The two team developers and I discussed how things were going and we agreed that I needed to find a way to insert my testing activities much earlier into the development of the sprint if we were to get the coverage needed and be able to test, find, and fix issues before the end of the sprint. We also agreed that we couldn't continue to accumulate technical testing debt in the form of manual regression tests.

Further research into agile testing techniques revealed some critical practices my team would need to implement in order to start using agile testing techniques. The most significant practices identified are listed below:

1. **Define and execute "just-enough" acceptance tests [1] -** This practice allows the customer to define external quality for the team and gives everyone confidence that user stories are complete and functional at the end of the sprint.
2. **Automate as close to 100% of the acceptance tests as possible [2] -** This practice prevents accumulation of technical testing debt in the form of an ever-growing manual regression test set that requires the team to stop and run the tests.
3. **Automate acceptance tests using a "subcutaneous" test approach with a xUnit test framework [2] -** Using an xUnit type framework and our software Application Programmer Interface (API) to automate acceptance tests allows for less-brittle test creation and easier development and

maintenance of an automated regression suit. This is compared to Graphical User Interface (GUI) test automation applications.

4. **Run all acceptance tests in the regression test suite with the build, daily (at a minimum) [4] -** This practice provides rapid feedback to the team if existing functionality has regressed by new code development changes.
5. **Develop unit tests for all new code during a sprint [5] -** This practice raises internal quality of the software and permits the "just-enough" acceptance testing described in number 1 above.
6. **Run all unit tests with every build [4] -** This practice provides rapid feedback to the team if regressions at the unit level occur with any code changes.
7. **Run multiple builds per day [4] -** This practice allows testing and exercising the latest code and changes throughout the day. It also allows for more frequent integration of developer code and thus quicker feedback into potential integration issues.

Again, these practices are what my team initially decided we needed to adopt in order to test early and test often, enabling us to find bugs in-line with development. This would also allow us to fix bugs at a cheaper cost before the developers moved on to another development task during the course of a sprint or project. But we had an immediate problem to address.

2. The Problem

The main problem was that our team didn't have an automation framework of any kind in place to implement several of the practices that would allow us to test in an agile way. Further, as we identified and discussed specific practices we were reminded of other areas of technical debt our team carried such as our manual build process. This process required three passes to build without error and no unit and functional test automation existed. These were problems we needed to contend with in order to begin implementing the practices we had identified.

3. The Solution

I discovered, through conversations with other agile test practitioners and additional research, a common element among teams already successfully implementing agile testing techniques: continuous integration. A continuous integration implementation seemed to be the solution to our lack of an automation framework. Now we needed to find out more about continuous integration so we could build our system.

4. What is Continuous Integration?

"Continuous Integration" describes a set of software engineering practices that speed up the delivery of software by decreasing integration times. It emerged in the Extreme Programming (XP) community, and XP advocates Martin Fowler and Kent Beck first wrote about continuous integration eight years ago [3].

Martin Fowler defines continuous integration as: *"a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible." [4].*

What does a continuous integration implementation look like?[1] Typically a continuous integration framework provides for automated source repository change detection. When changes to the repository are detected (e.g. when developers check in new code) a potential chain of events is put into motion. A typical first step in this chain of events is to get the latest source code and compile it (or for interpreted code perform some other checks like applying a form of lint). If compilation does not fail, then unit tests are executed. If unit testing does not fail, the application is deployed to a test environment where automated acceptance tests can be executed. If automated acceptance tests do not fail, the build is published to a public location for the team. The team is then notified (e.g. via email or RSS) and a report generated for the activities that included what and how many tests were run, the build number, links to results and the build, etc.

5. Continuous Integration Implementation

Our software development environment consisted basically of Windows® .NET C# applications. Thus several of our choices for implementing continuous integration were heavily influenced by what would work in this environment.

5.1. Layout

[1] To view a diagram of a possible continuous integration flow see "Figure 1, Continuous Integration Process Flow" posted at this URL: http://becomingagiletester.blogspot.com/2009/05/figure-1-continuous-integration-process.html.

The basic layout of our continuous integration implementation consisted of two physical machines and two virtual machines (VM). One physical machine hosted our virtual build machines. The other physical machine hosted our virtual test machines.

There is no reason that, given enough memory and CPU power on a single virtual host, we could not have chosen to host both build VMs and test VMs on the same physical machine. However we decided to separate the build VMs and test VMs to simplify management and restrictions in computing resources; our build VM host couldn't support the additional load of hosting our test VMs.

5.2. Software Tools Used

This is a list of the software tools used to implement our continuous integration.

- **Automated Build Studio** – Automated Build Studio™ (ABS) by AutomatedQA is an automated build and continuous integration Windows application. A license had been purchased for this application and was already being used to run manual builds on a few projects. Thus, while open source alternatives could have been used, we chose to continue using this tool as it was easy to configure and meant less tool-shift for our teams.
- **Software Test Automation Framework (STAF)** – STAF provides a service to send and receive remote commands between build and test machines such as copying files, executing programs, getting directory listings, etc. This tool was chosen as tests showed it to be a fairly robust way to handle communication between machines. The fact that it is a mature (seven-years-old) and open-source project developed by an IBM group helped us make the decision.
- **Visual Studio 2008** – Microsoft® Visual Studio 2008® development IDE includes a compiler that was already being used to build our applications.
- **Surround** – Seapine Surround SCM™ repository and versioning system is a source control system was already in place.
- **VBScript** –Microsoft VBScript™ was used to write a helper script to reset the test VM just before each new application install and test iteration. VBScript is natively supported on Windows platforms and thus is a good boot-strap language for Windows.
- **C# custom helper applications** – C# was used to create a few tools used to distribute, execute, and collect unit and acceptance test results. C# was chosen because we were most familiar with it (we were using it to develop most of our products), it's very powerful, and the .NET environment required to run C# applications was already installed on the build machine where they would need to run. The custom helper applications we wrote and used were:
    - **Test Distributor** discovers all NUnit project files in a specified product directory tree, and then copy all necessary .dlls and other files used by the project over to the product installation directory on the test VM for execution later
    - **Test Runner** runs each NUnit project in the product installation directory of the test VM.
    - **Test Results Processor** processes XML test results for each NUnit project test run, aggregates results (summary, list of failures, failure details) and write to html for later inclusion in build email.
- **NUnit** – NUnit is a test framework for all .Net languages used to execute unit and acceptance tests. We chose NUnit because the developers were already using it to develop unit tests and we also found that we could use it to develop and execute acceptance tests as well.

5.3. Tool Mapping

Here is how our tool set mapped to the layout described earlier:

1. Build VM
    a. Automated Build Studio
    b. STAF
    c. Visual Studio 2008
    d. Surround
    e. NUnit
    f. VBScript
    g. C# helper apps
2. Test VM Host
    a. STAF
    b. Virtual Server 2005
    c. VBScript
3. Test VM
    a. STAF
    b. NUnit

5.4. Putting It All Together

Our continuous integration implementation works like this[2]:

[2] Diagram of our implementation available at this URL: http://becomingagiletester.blogspot.com/2009/05/figure-1-continuous-integration-process.html.

1. The ABS service runs polling the Surround source repository for changes/check-ins
2. If a change is detected, a new build is started performing the following actions:
   a. Refresh source code on build machine for project/solution to be built
   b. Build the application (ABS using VS2008)
   c. Prep the test VM for product installation and testing; reset virtual machine and ping until it comes back online (VBScript helper script called via STAF)
   d. Copy installation files to test VM (STAF)
   e. Install application-under-test on test VM (STAF)
   f. Discover and copy all Unit and Acceptance tests to test VM (custom C# helper application, uses STAF)
   g. Execute tests on test VM (custom C# helper application, uses STAF)
   h. Copy test result .xml file from test VM to build machine (STAF)
   i. Process test results into results email
   j. Send email with test results (PASS or FAIL with details of failures), link to location of build, and build logs

6. Assessing the Team's Continuous Integration Practices

It's worth noting that Martin Fowler has put forth 10 *Practices of Continuous Integration.[4]* These practices help make continuous integration implementations go smoothly. Or put another way, trying to implement a continuous integration system without these practices could prove to be a rocky experience. The following list compares these practices with where our team was before and after we implemented continuous integration, providing a pre- and post-assessment of the practices.

1. **Maintain a Single Source Repository**
   ✓ Before CI: Seapine Surround SCM source repository
   ✓ After CI: No change
2. **Automate the Build**
   ✗ Before CI: Some partial automation; still very manual. Heterogeneous build environments
   ✓ After CI: Yes, with Automated Build Studio (ABS)
3. **Make Your Build Self-Testing**
   ✗ Before CI: Not self testing
   ✓ After CI: NUnit framework unit and acceptance tests
4. **Everyone Commits Every Day**
   ? Before CI: Unknown, varied probably
   ? After CI: We can only hope
5. **Every Commit Should Build the Mainline on an Integration Machine**
   ✗ Before CI: No, was not happening
   ✓ Yes, ABS's Continuous Integration Tasks helped us do this
6. **Keep the Build Fast**
   ✗ Before CI: Multi-pass builds, unordered dependencies
   ✓ After CI: 10 – 15 min; refactoring needed sooner than later
7. **Test in a Clone of the Production Environment**
   ✓ Before CI: Yes, but not automated
   ✓ After CI: Using clean virtual machine test clients to install and test
8. **Make it Easy for Anyone to Get the Latest Executable**
   ✗ Before CI: Not all projects using the common build repository. Some private file share locations for production code
   ✓ After CI: All products building to common location now. Build mail contains link to new build location
9. **Everyone can see what's happening**
   ✗ Before CI: Limited to ad-hoc emails, no web, no reporting, different project worked differently
   ✓ After CI: Use ABS's web interface to see the progress of builds, and email build and test status
10. **Automate Deployment**
   ✗ Before CI: Not being done
   ✓ After CI: Yes, automatically deploy the build, then test it

In summary, all continuous integration practices were either maintained, if existing, or improved upon.

7. Assessing Our New Agile Testing Practices and Agile Testing Capabilities

With our continuous integration system in place our team was now positioned to adopt all of the agile development techniques discussed earlier. And while not every practice relied on the continuous integration system, a few important ones did. These are identified in Table 3, Agile Practice Assessment after Implementing Continuous Integration.

Let's do a check up on where our team was now in regard to these agile practices. Practices marked with a ✓ were enabled using our new continuous integration system:

1. **Define and execute "just-enough" acceptance tests -** We made acceptance test definition a required task during sprint planning. The developers and customer both grew to like this as it gave them visibility into our test coverage and confidence that we were testing the right things.
2. **Automate as close to 100% of the acceptance tests as possible -** We now tried to automate as close to 100% acceptance tests as possible. But this process takes time to fully implement. There were still lots of legacy manual tests around but the awareness and commitment to automate them going forward were what we focused on. The net result was that we dramatically slowed our accumulation of technical test debt in the form of manual test cases.
3. **Automate acceptance tests using a "subcutaneous" test approach with an xUnit test framework -** We now used the NUnit test framework to automate our acceptance/functional testing using the API of the application. A nice side benefit is that our new tests going forward were all code, could be versioned in the repository, and could run in an automated fashion with the build.
4. ✓ **Run all acceptance tests in the regression test suite with the build, daily (at a minimum) -** Our NUnit acceptance tests now ran with a daily build.
5. **Develop unit tests for all new code during a sprint -** Developers started putting more emphasis on developing unit tests in NUnit for new code during the same sprint. This was a gradually improving process encouraged with the ease with which unit tests could now be run with the build.
6. ✓ **Run all unit tests with every build -** Our NUnit unit tests now ran with every build.
7. ✓ **Run multiple builds per day -** Builds were now started automatically when changes to the source repository were checked in. Manual builds could also be initiated.

As you can see in the list above, at least at a base level, all of the agile practices we set out to put in place. These practices started paying off immediately in terms of our ability to develop and test in parallel.

For example, now that unit tests were run with every build the team saw immediately when new code broke existing code. Moreover, I was able to immediately begin working with developers to automate acceptance tests as they were coding the stories. While the practices were still pretty new to us, it felt like a big win to have an environment that would allow us to improve how we worked together. We began finding bugs in the APIs I was using to automate the acceptance tests with. We also began finding other technical debt we needed to pay in order to keep going forward, like the lack of a command line installation for our applications or a programmatic way to call data validation to inputs that lived outside of the application GUI.

8. Retrospective

The journey from recognizing we needed to test in a different way to accommodate agile development to implementing the agile practices and continuous integration that would support it has been educational and rewarding.

First, learning how to change how I tested to operate in sync with developers was one of the biggest discoveries I've made in my testing career. Testers, developers, and all other stakeholders I've worked with have always wanted to be able to do this, but it wasn't exactly clear how to do it. Testing in parallel with development has overcome the traditional disjointed relationship between test and development.

Second, embracing the idea that we need to fully automate acceptance tests seemed both exciting and intimidating. The idea was very appealing but I also worried that it would be too difficult to scale. The subcutaneous test approach using xUnit test frameworks was the answer that finally made sense and seemed doable. Of course it required that I code much more and work with the developers to learn the API of our product.

Third, it's not hard to convince developers that automating tests is a good idea. But it was hard to convince them that we needed to go through our implementation "hump of pain" to get the pieces in place that would allow us to have continuous integration. I worked on a small team and we didn't seem to have any "extra' time for me to work on the infrastructure we needed.

I ended up working on the proof-of-concept during my lunch, sometimes in the evening or weekend, and during other down times. When I finally got things to a point where builds were automatically kicking off and the test VM launching, developers became engaged and excited. I identified the remaining tasks we would need to tie it all together. The team agreed to add them as sprint backlog items.

There is probably a more savvy way to approach the buy in for developing a continuous integration system, but I didn't know how else to approach it other than through prototype and demonstration. What I learned was developing a continuous integration system is as much the responsibility of the developers as it is the testers.

Finally, thinking of acceptance test development within the context of a continuous integration system has been a major shift. It's hard to think of going back to doing it the old way. I can't imagine working on a team that is not doing agile testing, let alone agile development. I, like many other agile testers, believe it makes much more sense and costs less to push testing and quality into the development cycle rather than to add it afterward. It adds value immediately. As I move to new teams, and begin working with them, my first step to implement agile testing approaches to accommodate agile development will be to consider implementing a continuous integration system to support it.

9. References

[1] "eXtreme Rules of the Road: How a tester can steer an eXtreme Programming project toward success", Lisa Crispin, STQE Jul/Aug 2001

[2] "Testing Extreme Programming", Lisa Crispin and Tip House, 2003, Addison Wesley

[3] "Continuous integration", http://en.wikipedia.org/wiki/Continuous_Integration

[4] "Continuous Integration", Martin Fowler, http://www.martinfowler.com/articles/continuousIntegration.html

[5] "Code the Unit Test First", http://www.extremeprogramming.org/rules/testfirst.html