How Not to do Agile Testing

Michael Puleio
Microsoft Corporation
mpuleio@microsoft.com

Abstract
When our team adopted Extreme Programming and Scrum, there were a number of challenges. Integrating an agile team in a “traditional” organization created friction and conflict within the team and with management. The team consisted of volunteers new to agile practices. We had difficulties with planning, estimation, task breakdown, managing requirements and working together in a collocated space. However, the biggest challenge the team encountered was not in any of these other areas, but in one that most of the team figured was simple: Testing. This paper describes how difficult it was to actually get the various testing practices in place and functioning properly. In particular, I discuss challenges we faced in coming to a common understanding of the practices, estimating testing effort, allocating the time required to do automated tests in a feature-driven culture, and getting a team of diverse individuals to understand how to apply principles that sound great but are challenging to implement.

1. Introduction

In March 2005, several individuals within the Windows Live/MSN Internet Access division were asked to participate in a new project. The project itself was to replace an existing legacy service with a new, extensible platform. This project was different than the usual project within MSN, as there was a secondary objective from the executive sponsor of the project to experiment with a number of agile techniques and methodologies and determine which might work well within the organization. The executive sponsor asked that we share our experiences, best practices, and failures with the rest of the organization. The executive sponsor asked that we share our experiences, best practices, and failures with the rest of the organization. After a lengthy negotiation, we chose to use Scrum for project management. We also adopted Test Driven Development, paired programming, and collocated workspace for engineering practices. We soon realized we were actually wrapping most of the engineering practices from Extreme Programming (XP) within Scrum. With extra discipline and effort, we could completely adopt Extreme Programming. At this point, we decided to take a leap of faith and adopt all of the principles of XP.

1.1. The Team

The core team consisted of one project manager, two software engineers, and two test engineers. For brief times in the project, there were three developers and three test engineers, but the additional team members were transient. Everyone on the team had between seven and ten years of experience building software applications and/or services. The team members had not worked together in a project previously.

The project manager was the customer representative, or Product Owner. Most of the project manager’s time was spent working with stakeholders, managing and clarifying requirements, and doing other Product Owner tasks. The project manager would spend the remainder of his time working as a team member. At times someone from outside the team would lend a hand and fill the role of ScrumMaster.

Initially, we had an outside mentor accept the role of ScrumMaster. After the first sprint, this mentor was reassigned due to a reorganization. At this point, the project manager accepted this role in addition to the Product Owner role. After one sprint with the project manager bending the rules of Scrum, the team decided that someone from outside of the team would be a better choice for ScrumMaster. This was a part-time role for one of the development leads for several sprints, until his schedule would no longer allow it. The team then went through three ScrumMasters in two months. They kept quitting, with the reason, “There is nothing for me to do. The team is able to handle everything without me.”

The team decided to follow the recommendations of Scrum: there are no set roles on the team. All team members pitched in to get the job done. This meant that developers learned how to test, and testers helped...
write mainline system code. Our PM, with no development experience, occasionally helped with both development and test work when his duties as Product Owner allowed him the time.

The team was lucky to receive limited mentorship from Ward Cunningham and Jim Newkirk, as well as more extensive help from Lakshmi Thanu and David Lavigne (our very first ScrumMaster). Ward taught us the philosophy behind the principles of agile development. Jim was able to help us with a number of challenges we encountered adopting test driven development and helped explain the philosophy behind TDD to the entire team.

1.2. The Project

The team was chartered to build a mission-critical, robust, reliable, extensible, scalable online service with 24x7x365 uptime. This project was a re-write of a business critical system for MSN Internet Access that handles a sustained average of several hundred transactions per second.

Unmanaged C/C++, C#, ASP.NET XML Web Services, and Windows services were used to communicate with a Microsoft SQL Server backend and Microsoft Passport. The entire system runs on Microsoft Windows Server 2003. The project was considered a medium sized project for our division. In the course of development, the team generated system code, unit tests, mock objects and system components, and test tools.

Our project was the first project inside our mid-sized group with executive sponsorship in support of agile practices, and we were initially allowed to experiment without hard deadlines and release functionality only when it was ready to replace the legacy system. In return for the lack of hard deadlines, management expected the team to share its experiences, best practices, and knowledge within the division in presentations, discussions, and through coaching other teams as they adopted Scrum. Later, business priorities shifted (as they often do), and hard deadlines were given. The team rose to the challenge and was able to deliver.

We experimented with a number of different ways of doing things. For example, we varied sprint length from 30 days down to 7 days, and eventually adopted 14 day sprints. We tried Promiscuous Pairing. We paired developers and testers together on both development and testing tasks. We also learned a number of things that did not work for our team.

1.3 Sharing Our Experiences

This paper is one team member’s views on how we initially failed at, and then became proficient at, testing in an agile manner. To illustrate the experiences of the team, I will use dialogs and conversations in this paper. The conversations are recreated from memory, notes, and product backlogs. They have been heavily edited and condensed to cover the topics of this paper. What appears as a few lines is the condensed conversation from long discussions. Also, I have not used any names in the dialogs, just Project Manager (PM), Developer, and Tester. This is to protect the innocent and the guilty. This also allows me to share both good and bad experiences without any team members being singled out. In a few cases, the dialog has been re-written, mixing up who said which lines. Finally, these are based on my recollections; others on the team may disagree with how events transpired and the gist of the conversations.

2. Challenges

The biggest area of contention the team encountered was testing. Neither the Product Owner nor the developers on the team understood the challenges and difficulty inherent in testing a complex software system in an automated manner. There were several areas regarding testing that we had challenges with: communication, estimation, and automation.

2.1. Communication

When five people from different backgrounds are pushed into working together for the first time, and in a small space, there inevitably will be some friction. In our team, there was a lot of friction and occasionally, fireworks. Much of the friction was caused by adapting to and learning a new way of writing software. Experimenting with practices, (changing sprint durations, trying promiscuous pairing, etc) also contributed to the occasional fireworks show. Finally, some of this was related to clashes between strong personalities.

When the project started, the project manager on the team barely understood the basics of development and testing; the developers knew almost nothing about testing and project management; and the testers were fuzzy on the details of professional development work and project management. Add to this the stresses involved in learning a new way of doing our jobs, and working in a small, enclosed space, and sparks flew. At first, due to the lack of common ground, working together was very stressful. Our communication skills needed serious work.
According to the Scrum guidelines, the tasks that a team commits to should be small, between 4 and 16 hours of work [1]. As a team, we targeted 12 hours as an upper limit. During our first sprint planning meeting this principle caused friction. For a new feature in the system, the test engineers on the team asked for 80 hours for testing. There was a lot of discussion around this estimate, which can be tied to a need for improved communication skills.

Project Manager (PM): “Why do you need 80 hours to test this feature?”
Tester: “Usually, testing takes at least as long as development. That is how we are basing our estimates.”
PM: “Can you break down the estimate into smaller pieces?”
Tester: “No.”
PM: “Are you sure?”
Tester: “You just don’t understand.”
PM: “No, I don’t. Please explain it in a way I can understand.”

Of course, as this continued, tensions rose, as did the volume of the discussion, until another team member was able to be heard (sometimes by yelling loudly) and called for a short break to allow everyone to cool down. (Believe it or not, this happened with surprising regularity for several sprints.) The participants were doing their best to communicate without a common knowledge base.

Another snippet of conversation was around the types of testing that needed to be done.

PM: “What do you mean you need 40 hours for exploratory testing in this sprint?”
Tester: “You asked for our testing estimates, and I have given them as a professional test engineer.”
PM: “OK, let’s try this differently. What is exploratory testing?”
Tester: “I can’t explain it.”
Developer: “Can you try?”
… (After 30 minutes of intense dialog)…
Tester (loud and very frustrated): “Ok, one more time: Functional testing ensures the system does what is expected under normal situations and inputs. Performance testing makes sure the system is fast enough. Stress testing ensures that the system can handle more than the expected load and finds the breaking point, so our Operations team can plan scaling the system. Exploratory testing is trying to determine what the system will do when it is used in a different way than what was intended. It may or may not lead to discovering bugs.”

PM: “OK, I think I understand. Sort of.”
And then a small breakthrough happened.
Developer: “Are there any books on testing you can recommend so we can all understand this a bit better?”
Tester: “Yes. I’ll get you a list to read. Can we drop this for now?”

Over the next month, everyone on the team read books about the other disciplines. The reading and discussion of these books created common terminology and understanding of basic principles. This foundation of knowledge, coupled with the experiences gained by actually doing cross-discipline work, taught everyone a common language. As a result, communication became much less strained, more positive, and work became enjoyable again. This increase in the ease of communication required a lot of hard work, time, and dedication on everyone’s part. It also helped increase the respect the team members have for one another.

Here is another situation in which our Product Owner pushed for more information from the team about the time investment needed for testing.

PM: “What value will I get for the investment of 40 hours of your time spent on exploratory testing?”
Tester: “Quality. You will be relatively certain that the system should work the way it is supposed to.”
PM: “That is a pretty vague response. Will I get at least a few bugs?”
Tester: “Not necessarily.”
PM: “Then why should I invest the team’s time on it?”
Tester: “Because it needs to be done. We cannot sign off on the system until it is done.”
PM: “Can this be done in a later sprint?”
Tester: “We need to do it. If we put it off, we cannot really sign off on the stories and features we want to get done in this sprint. None of the features can be marked as ‘Done.’”
PM: “So will this really take 40 hours?”
Tester: “Why do we have to justify our estimates when the developers don’t need to?”
Developer: “What do you mean; we don’t need to justify our estimates? We are breaking our work down as well as we can, and we can explain what value he gets for each piece of the system we build. I think that if you can explain testing in terms of value received for the investment of time, our PM will stop asking these tough questions.”
Communication in all forms, especially communication around testing, caused a number of challenges. However, once everyone had a common understanding, the improved communication helped all aspects of the project.

### 2.2. Estimation

Estimation is challenging. Most developers are quite poor at estimating work, usually under-estimating the time required to complete development of a set of features. This usually leaves little time for testing. Our team was no different than the norm. At first, our estimates were off by 50% to 100% or more.

Here is another perspective on a conversation that was previously mentioned.

**Project Manager (PM):** “Why do you need 80 hours to test this feature?”
**Tester:** “Usually, testing takes at least as long as development. That is how we are basing our estimates.”
**PM:** “Can you break down the estimate into smaller pieces?”
**Tester:** “No.”
**PM:** “Are you sure?”
**Tester:** “You just don’t understand.”
**PM:** “No, I don’t. Please explain it in a way I can understand.”

The big challenge here was a lack of experience in decomposing big tasks into little ones. Over the next three months, we gained experience. We learned that when we decompose a feature into development tasks, there were often many more tasks than we originally thought. We became much more careful and methodical in the task breakdown, looking for the “gotchas”. This did not stop the occasional setback, but it did help.

The tester in the above conversation did not, and still does not, believe that testing can be broken down into small estimable tasks like development work. In a recent conversation on this topic, he compared testing to the improbable act of removing cat hair from one’s clothes. Imagine, standing in your foyer, preparing to leave for a formal evening out. Your cat prances over and rubs itself against your legs. Now, like most cats, yours sheds fur in copious quantities. You only have a few minutes before you need to leave, so you run into the laundry room, and grab a lint brush. The lint brush (automated unit tests and acceptance tests) removes a good bit of the cat hair (bugs), but not all of it. Time is ticking away, so you start picking the hair off one strand at a time (running manual tests). Then the cat rubs back up against you (It’s a new code drop from the dev team). So, you firmly push the cat away with your foot and try the lint brush again. You know you will not have time to get all of the fur, but you spend the last few moments you have trying your best to quickly clean up (the ship date is approaching). You finally head out for your night on the town (you ship the software). During your evening out, someone notices some cat fur on the cuff of your coat (oh no! a bug was found by someone outside the development team, maybe even a customer).

**PM:** “So what will it take to test this new feature?”
**Tester:** “At least 30 hours.”
**PM:** “Come on. What about trying to break work into 8 to 12 hour chunks?”
**Developer:** “You know that our estimates get pretty bad for tasks that are more than 16 hours. How can we break this down?”
**PM & Tester:** “I don’t know.”
**Developer:** “How about we spend 30 or 45 minutes brainstorming out all the test cases that need to be written. We can then make estimates for each test case, and sum it up.”
**Tester:** “Good idea.”

[45 long minutes later…]

**Tester:** “There are about 25 test cases. Each will take about an hour to create. We also need to add functionality to the test automation toolkit, which should take about 16 hours. Let’s call it 40 hours total.”

**PM:** “Great. Let’s put these on the backlog as separate items so we can track them separately.”

What changed between these conversations? Three things: increased communication, more experience decomposing features into both development and test tasks, and a concerted effort by the entire team to figure out a solution to the problem. This experience was earned through making a number of mistakes and under-estimating a number of tasks. However, these mistakes allowed us to look back at our sprint backlogs and make incremental improvements to our estimates.

There were a number of times during a sprint planning meeting when a team member would provide an estimate, and someone else would recommend doubling the estimate based on the time taken in a previous sprint on a task with similar difficulty or scale. This sort of teamwork took a while to develop, but when it finally clicked, the team was able to get a lot done in a sprint. One of the testers does not agree with this breakdown because there are more things in testing than the test cases you uncover in a short brainstorming session. However, this strategy gives
everyone on the team a way to start finding realistic estimates for testing tasks.

2.3. Automation

There were also questions centered on test tools and test automation. After discovering the book Testing Extreme Programming [2], and getting everyone on the team to read the book, one of our often quoted passages was Chapter 19, which is best summarized as “No manual tests.” Manual testing slows down regression testing and the feedback loop. For example, with completely manual tests, it takes 5 days (1/6 of a traditional 30-day sprint) to do functional and acceptance testing. For the next three sprints, new features are added, each taking 5 days to test. As sprint five is started, 2/3 of the team’s time will be devoted to regression testing, leaving very little time for development work. With good, solid automated functional and acceptance tests, it should only take minutes to determine if there are any regression bugs in the existing functionality, leaving much more time for working on other aspects of testing.

There was a very simple tool that had been created for the legacy version of our project. This tool allowed very simple scripting to be done, but it required a lot of manual setup and interaction from test engineers. For early testing of the service, this tool would be sufficient. However, for testing features that were planned to be added later, this tool was insufficient and would require serious re-working, if not a complete re-write. Also, this test tool required a lot of interaction and manual effort to determine if a particular test passed or failed.

The test engineers on the team pushed hard for writing a new automatable, scriptable tool early in the development process. New functionality could be added to the tool in an iterative fashion during development (consistent with the spirit of Chapter 19). The Product Owner was initially opposed to the idea. His argument was that we had a tool that could get the job done for now, and would allow for more effort to be devoted to feature development rather than test tool development. Test pushed back against this decision, but was over-ruled by the Product Owner and developers. As a result, over time, the team accumulated what we termed “test debt”, or test items that needed to be done, but were deferred with preference being given to more features in the system. Finally, outside of normal work time, a few team members created independent tools that allowed scripting, automation, and more flexibility for test generation and execution. One of these was finally adopted by the team, after the Product Owner was convinced that a new feature could not be tested without it.

From this point forward, testing became a little bit easier. We had a tool that was easy to add functionality to, generate new test scripts in, and run for days at a time without any interaction. The test failures were logged in a way that described both the expected and actual results, so bug lists could be easily created from a test run log.

Not long after this tool was adopted, the team convinced the Product Owner that the test debt that had been accumulated must be taken care of before further development could be done. While no one was completely happy with this, everyone pitched in. Over the course of three 2-week Sprints, the team completely listed out and removed the accumulated test debt from the backlog. This required a shift in focus from approximately a 70/30 split between development work and test work in a sprint to a split of approximately 10/90. There were a number of bugs that were found in the course of these two sprints, and the fixes accounted for most of the development work.

Testing was never completely automated. Someone still needed to deploy the services, start the tests, and monitor the tests over long running times. However, this was such a large improvement over testing using the legacy test tool that we were able to streamline and shorten test cycles by around 50% and increase test coverage of the services.

We found that if automation of acceptance testing will be part of a sprint definition and the “done” criteria, and test automation is beyond the skill of the testers, the team must develop a test strategy and automation in advance of development (or at least in parallel). This can keep the feedback loop in sync with development so that testing basic functionality takes almost no time. If the feedback loop is not in sync with development there could be confusion about whether or not a task is “done.”

2.4. General Testing

There were a number of questions that arose in the course of the project. We were able to eventually find answers that worked for us. However, the answers evolved over time, and cannot be tied to any single, specific experience.

• How can a feature be coded and tested in a single iteration (e.g. four weeks, or even two weeks)?
• Is there a time when the developers “hand-off” the code and declare a code freeze to give the testers a chance to test the final system?
These questions are actually closely related. We found that the best way to ensure that a feature could be coded and tested in a sprint was to write the functional tests first, then develop the feature. This required negotiation between all the disciplines to ensure things worked smoothly. We did not follow this principle all the time. In fact, we did so only in a few sprints. In these selected sprints, development progressed very well, and there was no question when development of a feature was completed.

- When do performance testing, stress testing, and week-long “bakes” of the system occur in an iteration?
- How can you run regression tests, new functional tests, security tests, etc. every sprint without increasing the test window from a week to three weeks?

Test automation covers regression and functional tests. If your tests can be setup to run quickly on demand and/or overnight, regression testing and functional testing can be done every day or even every check-in. For the other types of testing, we need to look at one of the principles shared by Scrum and XP: potentially shippable code at the end of every sprint or iteration. In the case of an online service which requires extensive performance and stress testing, in addition to the normal functional testing, this is not always possible. The way we handled these types of tests, after much discussion and trying several other ways, was to stagger testing. For example, at the very end of Sprint 5, a performance test was setup to run in an isolated environment through the weekend. In the Sprint 6 planning meeting, we added tasks to the sprint backlog tasks to monitor the stress and performance tests for deliverables from Sprint 5. This monitoring usually entailed about 15 minutes of work every day for as long as the testing needed to continue. If no major issue was found during a week-long “bake” of the system under load, then the deliverables could be deployed. While this does not completely fit the recommendations and principles for Scrum and XP, it worked for us.

3. Summary

In the end, the project was successful. Even with the challenges we encountered, the team determined that testing can be done in an agile way. In fact, most of the team members have gone from skeptics of agile development and testing to evangelists. Those who are not evangelists do see value in the ideas and principles.

This project was a success in a number of ways. It allowed our team to learn a lot through trial and error. We were able to share these experiences with other teams internally. MSNIA now uses Scrum almost exclusively for project management. Our experiences, mistakes, and insights helped a number of teams when they adopted Scrum. Test Driven Development has been adopted by a few developers outside of this project team, and has been gaining support and momentum across the organization. This project had an extremely low number of bugs, when compared to other projects within MSNIA. Early on in the project, one of the testers made a remark to management during a sprint review, “This is the highest quality code that I have ever seen from this organization.” Even with all the setbacks, challenges, and a few bugs, the tester still believes in this statement today.

There were a number of other lessons learned by these experiences, which included:

- Communication is the key, no matter what you are doing. If you cannot find a common language and find that no matter how many words leave your mouth, others on the team cannot understand what you are saying; try to find a way to share enough experiences and knowledge from your background to communicate effectively. Our team found that books were one solid way to build this understanding, as was trying to do the work that is usually divided up to the other disciplines.
- Automation is the key to agile testing. Even if all you do for your first sprint is implement one feature for the customer, but you create a test automation framework you can extend easily over time, you will have made an investment that will pay off later in the project. This framework and tools should be up to your quality standards for production code.
- Estimating is tough, but it can be done fairly well. Estimating testing for agile development need not be too difficult. Techniques like Wideband Delphi estimation help, as does listening to each other, and learning from your mistakes (which you will make; no one is perfect)
- When the entire team is on-board and understands the difficulties of the other disciplines, solutions can be found.

These experiences have shaped not just this team, but other teams within Microsoft that are attempting to apply different agile methodologies and look to other groups for “Best Practices.”

4. Acknowledgements

This paper took a lot of effort from a number of folks. First, thanks to Robin, my wife, first-tier proofreader, and sounding board for ideas. Thanks to Barg Upender, my shepherd for this paper. Also, thanks to the team Bartholomew Hsu, John Boal, Mitch Lacey, Donavan Hoepecke, Mon Leelaphisut, Michael Corrigan (our initial executive sponsor), Jay
Schaffner, Kirk Brackebusch, Julie Nierenberg, Cale Carter, and Chris Burroughs, and everyone else in MSN Internet Access that helped out. Finally thanks to our mentors Ward Cunningham, Jim Newkirk, Lakshmi Thanu, and David Lavigne.

5. References


