

Agility for Testers

Elisabeth Hendrickson

Pacific Northwest Software Quality Conference

October 2004

Background

“Agile” is a great buzzword. In everyday language, “agile” means adaptable or able to move or respond quickly. Given the pace of software development today, everyone wants to be agile.

But it’s more than a buzzword. The phrase “Agile Methods” has become an umbrella term for a collection of methodologies that increase agility, including Extreme Programming (XP), Scrum, Crystal, and Lean Development. The Agile Manifesto describes the values of the Agile community: individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; and responding to change over following a plan.¹(The Agile Alliance, 2001)

Agile teams accept change as inevitable and tailor their processes accordingly. Short iterations mean that stakeholders can see steady progress and provide frequent feedback. Continuous integration means that if one part of the system isn’t playing nicely with others, the team will find out almost immediately. Merciless refactoring, where programmers improve the code internally without changing its external behavior, prevents code from becoming fragile over time. Extensive automated unit tests ensure that fixing one bug won’t introduce two more.

Agile teams test early, often, and relentlessly. Many Agile teams perform extensive unit testing and collaborate with users on creating automated acceptance tests. Some teams even write automated unit tests before writing the code those tests will exercise. Agile programmers hold themselves accountable for the quality of the code, and therefore view testing as a core part of software development, not a separate process performed at the end.

The original founders of the Agile community are programmers and the early literature had little to say about the role of independent testers. In fact, just a few years ago, some Agile proponents optimistically suggested that diligent early unit testing and automated customer-driven acceptance testing reduced the need for independent system testers.

These days, many teams, including one I have been working with for several months, are discovering that testers can indeed add value. Skilled testers have an uncanny ability to expose the flaws in software in a surprisingly short amount of time. Programmers are

¹ These values are expressed in the Agile Manifesto, reproduced in its entirety including the copyright notice and author names at the end of this paper. The Agile Manifesto is also available from the Agile Manifesto website: <http://www.agilemanifesto.org>.

often dumbfounded by the speed with which a good tester can find a defect in supposedly completed code. “But I tested that!” they protest.

So the question is not, “Are testers needed on Agile projects?” but “How can testers contribute most significantly to Agile projects?”

In this paper, I contrast “Traditional”² and Agile testing, describing how independent testers can contribute effectively on Agile projects by shedding some practices that no longer fit and adopting others made possible in an Agile context.

What’s Really Different about Agile

Unfortunately, as often happens with great buzzwords, some teams claim they’re “going Agile” when all they’re really doing is compressing the schedule, tossing out the documentation, and coding up to the last minute. As Abraham Lincoln said, “If you call a tail a leg, how many legs does a dog have? Four. Because calling it a leg doesn’t make it a leg.” Similarly, calling a project or team Agile doesn’t make it so.

For teams that adopt Agile development practices, agility means delivering working code at frequent intervals. Each of the Agile methods has its own set of practices for doing that, but most seem to agree that short iterations, continuous integration, lots of team communication, and frequent feedback are critical. Agile methods spurn any activity that doesn’t add value to the final product. In lean manufacturing terms, activities that don’t contribute directly to the end product are considered waste. (Poppendieck & Poppendieck, 2003)

Agile teams are typically co-located in a team room where big, visible charts help convey information at a glance. Some use online collaboration tools like a Wiki (a web site with pages anyone could edit from their browser).³ All seek frequent feedback from the business stakeholders.

XP lays down the most rigorous set of development practices of the Agile methods. In XP, programmers pair up to write code so every line is inspected as it’s written. They develop the code test-first, so there is always a comprehensive set of automated unit tests that must pass before the code can be checked in.⁴ Both XP and Agile programmers are sometimes called “test-infected,” meaning that they’ve been infected with the idea that testing early and often (as opposed to debugging) will help them write better code.⁵

Agile developers have developed a shared vocabulary that expresses their programming ideals.⁶ They avoid “BDUF” (Big Design Up Front) and instead Do the Simplest Thing That Could Possibly Work. When someone suggests adding code “just in case,” they’re reminded “YAGNI” (You Aren’t Gonna Need It). The Once and Only Once (OAOO) principle aims at keeping duplication out of the code. This emphasis on simplicity is one

² The word “traditional” is somewhat ironic given the relative youth of the software industry. By traditional, I mean a formal test process that is focused on the independent verification and validation of the system under test.

³ For more information on Wikis, see <http://www.wiki.org>

⁴ For more information on XP, see Beck’s *Extreme Programming Explained*. (Beck, 1999)

⁵ As evidence of the growing popularity of testing among the programming community, there were several books on test driven development (TDD) published in the last couple of years.

⁶ A good place to start investigating Agile lingo is <http://c2.com/cgi/wiki?YouArentGonnaNeedIt>

of the things that sets Agile practices apart from traditional heavy weight and code and fix software development.

Perhaps more important than any single Agile practice is that within the Agile community, these practices are considered the norm. Agile programmers truly think differently about software development.

Conflicting Values: Traditional Testing and Agile Development

In traditional testing, system testing is usually done at the very end of the project and is often the first time all the components of the system are integrated together. The predictable outcome is that system testing reveals numerous issues that must be resolved before release. Furthermore, because the system testing occurs at the end of the development cycle and the release date is often fixed, the time for system testing is likely to be squeezed when development dates slip.

In order to resolve these conflicting forces—too many problems to find and not enough time in which to find them—formal system test and quality practices emphasize preparation, documentation, adherence to plan, clean handoffs, and strict change management.

These formal practices arose from contexts in which the team relies heavily on system-level testing performed by an independent test group as the last step in the development cycle.⁷ They make less sense for Agile teams with test-infected programmers.

Traditional Wisdom	Agile Perspectives
Strict change management	Change is inevitable.
Comprehensive documentation	Working software is more important. And face-to-face communication is better anyway.
Up front planning	Plan to the next iteration.
Formal entrance and exit criteria with signoffs	Collaborate, don't hand off.
Comprehensive system-level regression tests	Detect defects earlier with automated unit tests and continuous integration.

This conflict in values results in tension when testers attempt to apply the same tried and true techniques from heavyweight or code and fix contexts on Agile projects. As one programmer commented to me, “We’ve changed the way we do things completely, but our QA manager wants to keep doing the same old heavyweight things. They just don’t have any value any more.”

⁷ There is a growing movement within the testing community toward Context Driven testing. For more information, see the Context Driven Testing Wiki, <http://www.context-driven-testing.com/wiki/>

An Agile Testing Story

I recently had the good fortune to participate in three successive projects with an XP team. I was on the projects part time, a member of the development team but with a distinct role as a tester. This is unusual for an XP team where individual team members are expected to be generalists without specific roles. Both the team and I were experimenting with how a testing professional could add value on an XP team.

My first day on the project, one of the programmers helped me install the development tools and connect to the source control system. My task was to begin exploratory testing on the features that were completed.

A traditional QA department wouldn't have touched the application at this point because it was nowhere near ready for system test. But by testing so early, I found issues while there was still plenty of time to address them. This practice paid off almost immediately. One of the first bugs I found required not just a simple code change but also a design change.

It's worth noting that I could not have been as effective testing early if the programmers hadn't been writing the code test-first. I came into the project in the second week and was able to be productive executing tests immediately, but only because the software already behaved predictably under most conditions. If it had not been so clean, I would have spent most of my time attempting to get the system to work at all.

Agile Implications for Testing

Before joining the XP team, I already had plenty of evidence that testing software developed using Agile practices would be very different from testing software developed in a code-and-fix or process-heavy context. The good news is that the software is easier to test because it is more reliable and testable. But I discovered that Agile methods pose three key challenges for testing:

- The software is a moving target
- Short iterations and frequent releases increase time pressure
- The risks are shifting

Software as a Moving Target

In traditional testing, system testers typically push for an early code freeze so they have time to test the feature-complete system.

This simply isn't possible in an Agile context. There may be some short period of time in which new stories are not being implemented, but it is not the month or more that system test teams are accustomed to having for a full regression cycle.

Agile testing must take that continuous change into account. That means finding ways to test earlier.

Short Iterations

Traditional test teams accustomed to having 4 – 6 weeks at the end of a release cycle to do nothing but end-to-end system or regression tests will discover that simply isn't possible in an Agile context. The iterations are too short.

Agile testing must provide feedback not only sooner but faster. The solution to these first two challenges is not to find ways to do the same type of testing only faster, but to re-evaluate the testing processes entirely.

Shifting Risk Profiles

For testers accustomed to working in a code-and-fix context where any small change can have an unanticipated ripple effect through the rest of the system, the lack of time for full regression tests may be frustrating. They perceive any change to the system as introducing new risks. Fortunately, the risk that such a change could occur, while not gone altogether, is significantly less with an Agile project where the programmers have created extensive automated unit tests.

Agile projects still have risks, of course. Agile practices, particularly XP programming practices, mitigate risks associated with late breaking change. Other risks remain. The Agile tester's challenge is to detect those new risks and determine how to respond to them.

For example, I realized on the XP projects I worked on that we had significant risks around:

- The parts of the system that stayed mocked out the longest.
- Our assumptions about production data we didn't control.
- Areas of the system that didn't have many automated acceptance-level tests.

By understanding the risks, I was able to focus my test efforts more effectively. I found a number of issues by testing around each area where we interfaced to other software and data that we didn't control.

Another example of shifting risks: one QA group was using manual smoke tests as a way of ensuring they didn't waste time testing software that didn't meet a basic level of quality. In the past, about half the builds had been Dead on Arrival (DOA) in QA. A few months after the programmers switched over to XP, the QA group realized that the builds always passed the smoke tests. Now they were wasting time by smoke testing—the software just worked.

This QA group discovered that they no longer had to worry about the risk that the build would be broken. So they abandoned their manual smoke tests and focused on running more interesting, informative tests.

Toward More Agile Testing Practices

I discovered that while everything I already knew about testing helped me work with the XP team, I had to adjust my style in order to work more effectively with the team.

Offering Feedback

I provided feedback to the team in various forms: sometimes with a side-by-side demonstration while pairing, sometimes by adding a new story card, sometimes by making a note on the Wiki, sometimes by creating an automated test to demonstrate the problem, sometimes by entering a bug record in our Bugzilla database. The form of the feedback was less important than the content.

At first I was a bit uncomfortable with this free form flow of feedback. I was accustomed to capturing items in a bug tracking system. I missed the ability to search through a central database of issues to see what else had been found and to review the issues I'd raised already.

My discomfort remained until I realized that providing the information in the most usable form was the most important contribution I could make.

As an example, toward the end of the project I discovered that a series of searches didn't work. When I showed one of the programmers the issue, he said "Oh, I know what that is. Hang on while I fix it." In this case going through the effort of filing a bug would simply have increased the overhead of the communication enough additional benefit to justify the cost.

Avoid the Customer Surrogate Role

Our customer for this software was not working in the same office. He came in to work with the team once a week. This time onsite enabled us to demonstrate the software, get his feedback, and have him clarify stories. Sometimes I did the demonstration. Other times one of the programmers did.

Because our customer was offsite, there were times when the programmers asked me customer questions. "Should we make the error message look like this or that?" they'd query. It was tempting to slip into a customer surrogate role. Because I was not very familiar with the domain, I was able to resist temptation. It turned out to be a good thing: my natural inclination turned out to be wrong in most cases.

One of the key aspects of the Agile movement is that the business stakeholder is responsible for the business decisions.

Minimize Documentation

Test documentation can account for a large percentage of the test effort. I've been conducting informal polls in my classes to understand exactly how much time test documentation takes. 135 testers across 57 organizations revealed that testers spend about one third of their time just documenting test cases.⁸

Knowing the cost of heavyweight documentation and keenly aware of the tight timeframes, initially I resisted documenting my testing. But I learned on the first project that the rate of change actually made it more important that I document what had and had

⁸ It's worth noting that this is an unscientific poll where the data may be skewed by several biases, including selection bias (only people in my classes participated) and self-reporting bias (participants were reporting their time from memory). Despite the poll's flaws, the results are both interesting and consistent with my own experience working in document-heavy environments.

not been tested. As an example, at one point a programmer asked me, “Did you test for that?” I replied, “Yes” because I was sure I’d run that test sometime in the last few days. But the more important question was, “Have you tested for that since yesterday?” I would not have been able to answer that question.

Although we all realized that documenting my test effort was important, we didn’t want that documentation to become burdensome.

We were already using a Wiki, a web site with pages anyone could edit from their browser. The Wiki software we used has a table plugin we used to keep tabular data (like story status) straight⁹.

In order to track testing, I set up test status grids with brief test descriptions, date last executed, drop down lists for configuration (local v. staging), and test result (pass, fail, blocked). The result looked like this:

Test #	Category	Test Name	Last Executed	Result	Configuration
01	Home Page	All sections present	8/30/04	Pass	Staging
02	Search Results	Pagination Tests	8/30/04	Blocked	Staging
03	Search Results	Input Tests - special characters, etc.	8/30/04	Fail	Staging
04	Search Results	Ordering/Sort Tests	8/31/04	Pass	Staging
05	Search Results	No results returned	8/31/04	Pass	Staging
06	Contact Us	HTML Email	8/31/04	Pass	Local
07	Contact Us	Text-only Email	8/31/04	Pass	Local

The table control let me march down the table of test cases and select the values from the drop down. Tracking status became almost zero overhead, and everyone knew what I was testing and where.

Automated tests also become a source of documentation. They may even become the most up to date and accurate specification available. I recently needed to remember how some software I had tested worked. I was able to get the answer from the automated tests more quickly than from the official specification.

Sometimes the automated tests are not sufficient by themselves, but they can be made to generate the necessary documentation. In one case, we added logging that turned the automated test steps into human-readable documentation as the tests ran.

If you need additional documentation beyond what the automated tests provide, consider using a Wiki. It’s public, visible, open, and archivable.

Don’t Duplicate Automation

Although the team produced phenomenally good code, I was still finding an average of 5 issues a day. Many of these were small issues: typos or formatting discrepancies between our implementation and the customer’s mockups. Some were unanticipated requirements such as the need for exception handling when the user manually edited the URL and made it invalid in the process. A few were serious bugs resulting from integrating our software with an existing framework we didn’t control.

⁹ We were using Twiki (<http://www.twiki.org>) with the Table Plugin (<http://www.multieditsoftware.com/twiki/bin/view/TWiki/TablePlugin>)

Originally we thought it would make sense for me to write automated tests in jUnit with jWebUnit to demonstrate the bugs. Ultimately, we gave up on that idea: it made me too inefficient. It took me an average of about 10 minutes to log an issue as a bug or show it to a programmer. It took me half an hour to write the test. Since I was a part-time tester supporting a team of agile programmers, I couldn't afford the extra time.

I also tried to automate some of my end-to-end tests. We found that although my automated tests were a little different from what the programmers were already writing, they weren't different enough to justify the time I was spending on them.

We also discovered that we had some stumbling blocks attempting to automate tests to run in the fully deployed system (as opposed to the mocked out system). As an example, one of the bugs I found involved an email feature. Reproducing it involved setting a preference in another part of the system we didn't control, then forcing an email to be sent, then checking whether or not the email arrived. While it would have been theoretically possible to automate that, it was cost prohibitive: it took a few moments to verify manually as opposed to the several days it would have taken to get automation working with all the parts of the system I had to touch.

So we finally decided it made sense to let the programmers do what they do best (write code) while I did what I do best (test). The programmers continued to create unit tests test first. And in some cases, I paired with programmers to write acceptance-level automated tests. But for the most part, I focused on testing the software in ways they had not already.

Although I didn't automate as many end-to-end tests as I originally assumed I would, I did use scripts extensively to support my manual testing, particularly for data setup. When the application was in the early stages, there was no way to push data into the system without writing code to do so.

One alternative that we did not have the opportunity to try on these projects but that other Agile teams find works well for them: the open source test frameworks Fit and FitNesse¹⁰ provide enable testers and programmers to collaborate effectively on test automation. With these frameworks, programmers can write support code to connect the tests to the system while testers can contribute test cases very quickly by adding data to a table.

Use Exploratory Testing

Exploratory testing is a highly disciplined form of testing in which the tester is simultaneously learning about the program, designing tests, and executing them. This kind of testing involves using rigorous testing techniques just as pre-planned testing does, but applying those techniques directly on the software being tested instead of writing about it first.

On the Agile-Testing list, Ward Cunningham said "...agile programs are more subject to unintended consequences of choices simply because choices happen so much faster. This is where exploratory testing saves the day. Because the program always runs, it is always ready to be explored."(Cunningham, 2004)

¹⁰ See <http://fit.c2.com> and <http://www.fitnessse.org>

Integrate Testing into the Development Processes

In the last few months I discovered first hand what several Agile practitioners have been telling me for much longer: having everyone in one room speeds up communication by an astonishing amount. Questions that might otherwise take a couple days to be answered over email are answered within a couple of minutes. There were numerous occasions when I was able to save hours of my time by overhearing a conversation in the team room.

But I also learned that collocation is not sufficient to ensure that the testing is fully integrated with the efforts of the rest of the development team. Even though I was sitting right next to the programmers, there were occasions when I felt as though I was handed software over a wall.

In order to integrate my test efforts more completely with the programmers work, we learned to:

- Include test activities such as test data creation in each iteration's plan
- Share our test data—though not necessarily use the same test data
- Pair on creating test infrastructure code

The Evolving Role of Independent Testing

On traditional teams, testers often view themselves as a last line of defense. They may feel that they're protecting the unsuspecting users from the sub-standard software the developers would otherwise inflict on them. The result of this stance is usually some degree of tension between the testers and developers, where the degree of tension can range from mild to vitriolic.



Agile testing requires a new take on the role of the independent tester.

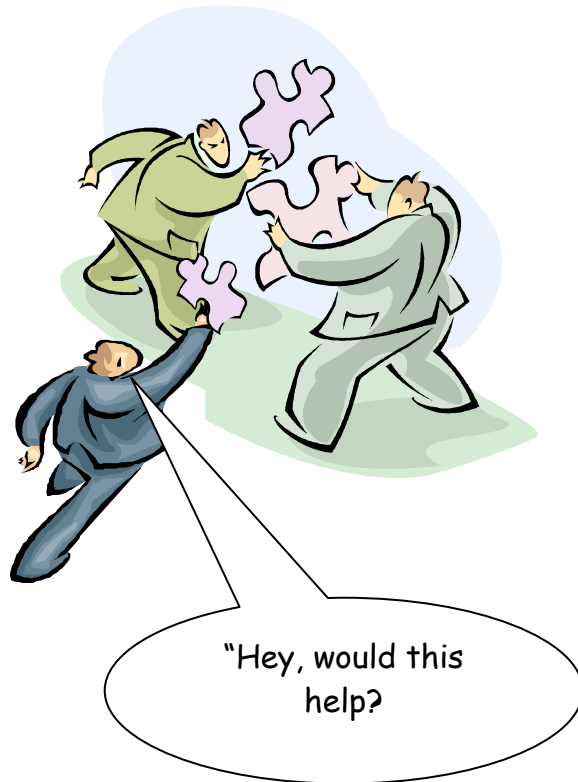
Testing cannot exist in a vacuum. Testing yields information about the system under test. But for whom? Who needs that information and what are they going to do with it?

In an Agile context there are two key sets of stakeholders who need the feedback that testing provides:

- The developers need information they can use to improve the code, including the unit tests.
- The business decision makers¹¹ need information they can use to guide the project.

Instead of taking the role of last line of defense between the users and the project team, testers on an Agile project do better to take a supporting role to these two sets of stakeholders.

¹¹ In XP, this is the Customer role.



A traditional view of testing suggests that independent testers exist to assess the quality of the system. The argument for independent assessors is that they won't have the blind spots that the developers have. They can objectively reconcile what the users or business stakeholders said they wanted (requirements) with what the developers delivered (implementation).

This independent assessment has less value for an Agile team working closely with a customer. The customer can assess for themselves whether or not the implementation meets his or her needs. Testers should focus on providing feedback and information rather than assessing quality, exposing bugs, or evaluating compliance. They may help facilitate discussions between programmers and business stakeholders.

Taking a Supporting Role

In addition to supplementing the team's test effort, Agile testers can support the team by:

- Asking "what if" questions of both the programmers and business stakeholders in the planning game. For example, "What if the migrated data has null values?" or "What if a user decides to...?"
- Analyzing risks and providing information early. For example, "A similar Web application that used a similar mechanism was hijacked by spammers. What safeguards do we have in place to ensure this won't become a spam machine?"
- Offering information about external dependencies or requirements that the team might not otherwise know about. For example, "I would expect this application to look and feel like this other related application that was just released. Will it?"

It's important to remember that testing is as much about perspective as skill. Where the programmers are thinking about things like whether or not to use the Singleton Pattern¹², testers are thinking about the likelihood of a NULL sneaking into the database. Programmers have their heads wrapped around the emerging design. Testers have their heads wrapped around the emerging risks.

Supporting, not Servile

Taking a supporting role doesn't mean serving as a personal assistant to individuals on the team, however.

In traditional contexts, programmers have sometimes asked me to test their code before they check in. I almost always refused their requests because it would double my effort. If I tested the code on their machine before it was checked in, I would still have to test it in the build.

In light of my recent experiences with XP, I've been wondering if I was doing the right thing. I realized that in some cases, I could have saved everyone a lot of time if I'd helped the programmer earlier, especially since the time delay between code and test can be days or weeks on a traditional project. But in other cases, I would have been supporting an individual programmer at the expense of the team.

Consider the difference between two requests by two programmers, both when I was an independent tester in a non-Agile context.

One programmer asked me to review his code and watch it execute at his desk, with him present, before he checked it in. This was several years ago, before I'd heard the term Pair Programming, but that's what he was looking for: a pair, another set of eyes. Back then, I hesitated before agreeing. Today I wouldn't hesitate at all. This is the kind of team support we all benefit from providing and requesting.

Another programmer tried to hand me a floppy disk, saying, "Hey, can you test this?" He wanted to hand off responsibility for testing to the nearest available tester so he could move on to other tasks. He had no idea whether the code he'd written was any good and he thought it was someone else's job to find out. I refused his request and learned later that he had to rewrite almost the whole thing because he hadn't understood the interfaces. Doing that programmer's testing for him would have been more like enabling than supporting—enabling him to continue to do a sloppy job.

Conclusion

If programmers change how they create the software, testing needs to change as well. Traditional test processes are not Agile: heavyweight documentation, fragile automation, and extensive test tracking slow down the test effort and make it more difficult for the test process to adapt to extensive changes in the software.

We can become more agile, and thus adapt better to Agile methods if we:

¹² Although not explicitly an Agile technique, Design Patterns are well known by the Agile community as a whole. In fact there's a great deal of overlap between the Agile community and the Patterns community. For more about patterns, see the now classic Gang of Four (GOF) book, (Gamma, Helm, Johnson, & Vlissides, 1995)

- Streamline our test processes
- Focus on providing feedback not assessments
- Shift our role from last line of defense to team supporter

Without these changes, testers find themselves at odds with Agile programmers. But by becoming more agile in our approach, testers can have a huge positive impact, helping Agile projects be even more agile.

The Agile Manifesto

From <http://www.agilemanifesto.org/>

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

© 2001, the above authors

this declaration may be freely copied in any form,
but only in its entirety through this notice.

End Notes and Resources

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*: Addison-Wesley.

Crispin, L., & House, T. (2002). *Testing Extreme Programming*: Addison-Wesley.

Cunningham, W. (2004). "Re: [agile-testing] Summary of Position". *Agile Testing list*
Available online: <http://groups.yahoo.com/group/agile-testing/message/3881>.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns*: Addison-Wesley.

Kaner, C., Bach, J., & Pettichord, B. (2001). *Lessons Learned in Software Testing*: Wiley.

Pettichord, B. (2003). "Where Are the Testers in XP?" Available online:
www.stickyminds.com/se/S6217.asp.

Poppendieck, M., & Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*: Addison Wesley.

The Agile Alliance. (2001). "The Agile Manifesto". Available online:
<http://www.agilemanifesto.org/>.

Acknowledgements

Many thanks to the following people for reviewing an early draft of this paper: Brian Marick, William Wake, Jonathan Kohl, Jeffrey Fredrick, Daniel Knierim, Marc Kellogg, Danny Faught, Ron Jeffries, Hubert Smits, Rob Mee, Sherry Erskine, Amy Jo Esser, Gunjan Doshi, Dave Liebreich, Janet Gregory, Chris McMahon