Testing in an agile environment

James Lyndsay, Workroom Productions Ltd.

Abstract

A practitioner's view of agility and testing. Starting with a thought experiment on the application of the Agile Manifesto to software testing, and moving on to a description of the effects on testing of typical practices found on agile projects. Examines ways that testers can bring value to agile projects, and typical problems encountered in agile environments. Concludes with a brief description of ways that a tester can facilitate learning.

Software testing and Agile Projects

The 'Agile Manifesto' arose from an informal gathering of thinkers in a ski lodge in early 2001 (see http://agilemanifesto.org/history.html). It has come to summarise the reasoning behind a collection of software development practices that try to provide an alternative to 'documentation driven, heavyweight software development processes'.

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.

This text is taken from http://agilemanifesto.org/ .

How does this manifesto affect testing? What would testing be like if we applied these values as a test strategy? What is it like to work as a tester on a project which has these values?

Agile values applied to Software Testing

Testing is a key element of software development. A product is tested so that it can be judged; for completeness, for its fitness for purpose, and for risks. It is vital because it produces information that feeds back into the creative process.

Testing differs from the design and creation of code because:

1) it does not directly produce something of *ongoing* value to the customer

2) it cannot be completed – especially in the search for surprises

We can work through the values in the Agile manifesto, and judge how each applies to the problem of software testing.

Value working software over comprehensive documentation

This position has a fundamental effect on software testing. Many test techniques require comprehensive documentation to allow effective test design.

Testers on an agile project need to be able to work with software products directly; reading unit tests, exploring a system, analysing a comprehensive range of output. Testers also

need to be able to interact directly with designers and coders to understand the technological imperatives and restrictions that affect the software and its unit tests. Conversations and shared understandings will take the place of some documentation.

It comes as no surprise to testers that *working* software is not the same as code – the tester clearly needs to be involved in not only assessing the product, but in deciding how the product is to be assessed. However, with automated unit tests in the hands of the coders, and confirmation-focussed acceptance testing driven by the customer, testers should be aware that they will not be the sole – or even the primary – owner of deciding what works, and what doesn't.

Value individuals and interactions over processes and tools

Testing will be driven by what is important to a user, rather than to fulfil a procedural requirement. Where a regulator imposes mandatory testing on a product, that regulator will be a key customer to the project. Where such testing is imposed by custom, that custom will be questioned.

It is better to have communication between tester, customer and designer than to maintain independence of the test team. The test team will have to weigh up the virtues of independence, and try to reach them in a different way, or to compensate for their absence.

Testers should be on their guard for the impact of test process and test tools, and try to avoid imposing rigidity or friction that hinders direct interactions.

Note: for testers, this could be read as favouring manual testing over automated testing. In practice, it is common to find large-scale automated unit testing on agile projects, to confirm that code works as expected. The product will be judged by the customer typically by manual, confirmatory tests, with close observation for undesirable behaviours. Testing by testers is often driven by the need to measure the system's performance and to find surprises – tools are very much in evidence, but rigid test scripts and procedures do not give the requisite opportunity for discovery, diagnosis and exploitation.

Value customer collaboration over contract negotiation

Testers are key collaborators with the customer, and on some agile projects will take on much of the role of the customer in designing and executing confirmation-driven acceptance tests. However, although testers traditionally make good customer advocates, working closely with a customer is preferable to becoming a proxy. The tester will frequently take on the role of extending a customer's tests to the limits of the application, acting in some cases as an undesirable user or 'Bad Customer'.

Test strategies which lean heavily on an unchanging set of requirements (for example: designing and coding tests to be bought together with code late in the project; prioritising tests based on a fixed risk assessment; testing only what has been agreed in the contract; reporting

bugs only against fixed requirements) may be considered to be fatally flawed in the light of this value. Iterative collaboration is favoured over a negotiated bill of work.

Value responding to change over following a plan

An acceptance of change is at the heart of agile projects – there is no requirements freeze before coding starts, and this can cause substantial problems for testers (and coders) used to working to a fixed set of requirements.

Large-scale automated unit testing gives a stability to the code and to the product that goes some way to reducing the effects of constant change, and testers need to be able to give this support where necessary.

While this idea has clear effects on testing, its direct application to testing is less clear-cut. Although the test effort will re-prioritise its work dependent on current values, in practice the teams on an agile project may *expect* testing to follow a plan – especially where one of the roles of testing is to work apparently untouched areas of the system to spot undesirable behaviours arising from recent changes. Test plans for work within an iteration (and in sequential iterations), are valuable to the ongoing stability of the product itself and can allow testing to give feedback across the system, rather than simply on currently-interesting areas.

Testing on an Agile project

While is may be interesting to consider how the Agile Manifesto might apply to testing in isolation, testing in practice is always part of a greater project. It is informative to look at the practices on an agile project, and to judge how they affect testing.

There are a number of different 'Agile' methodologies, each with its own combination of practices. None of the practices of Agile methodologies are unique, or novel; the strength of a methodology lies in the ways combinations of good practice are combined into a framework, rather than in the individual elements themselves.

For this paper, we will look at the practices of eXtreme Programming, and in particular the twelve practices listed by Kent Beck in his seminal book 'eXtreme Programming Explained'. XP shares many of its core practices with other agile methodologies, and is a good proxy for study.

Shared Roles

An element common to most agile approaches, but not noted explicitly in the XP-focussed set below, is the blurring of the distinctions between designers, coders and testers – and, indeed, between managers and workers, or between customers and developers. While individuals in an agile team may specialise in particular roles, most people will take on different roles depending on context. The tester who is out of their depth reading code or uncomfortable influencing design decisions will find themselves at a serious disadvantage on an agile team.

Automated testing is at the heart of agility

The vast majority of tests (by number and frequency) on an agile project are automated, and pass every time. Automated confirmatory tests are so fundamental that, without them, it is hard to call a project agile at all.

Comprehensive testing, but not by testers: *The code is written within a scaffolding of automated unit tests, and the functionality is judged to be complete when it passes the customer's confirmatory acceptance tests.*

Automated unit tests are not written by testers, but by coders. In Test Driven Development, the tests are written just before the code, executed once to fail as a control, then executed again after coding to show that the previously-failing test now passes. Typically tests focus on very small amounts of code – often one line of code is covered by multiple tests, written one by one.

While this may seem frustratingly slow, it produces accurate, elegant, modular code that is, in some sense, documented by its tests. Code produced with comprehensive unit tests is

tangibly different, from a tester's point of view. It typically exhibits far fewer failures in normal use. Bugs fixed tend to stay fixed, and fixes themselves introduce fewer new faults.

Test specialists can influence the scope of these unit tests, using their knowledge to help the coders consider the effect of conditions that may have been omitted when thinking only of the successful path. Testers may add value by browsing unit tests looking for gaps, or by introducing test design techniques.

The customer is responsible for tests that confirm that the working system delivers the desired capability. The test specialist can again help with these tests, by facilitating automation and the construction of test data, by improving test design, and by introducing analysis tools. In some projects, the tester becomes a substitute for the customer, taking on the tester's traditional role of customer advocate. Although this is common on projects where the customer is not available, where a potential customer has not been identified, or where there are many different customers, it is not an ideal situation; the tester may not notice important side-effects of functionality, or may wrongly assess new risks.

Some agile teams fall into the mistaken assumption that comprehensive, automated unit tests and comprehensive, automated customer acceptance tests are all that is required. However, both of these types of tests confirm that the system works as expected. They will only flag unexpected behaviours if those behaviours coincidentally cause a test to fail. Test specialists often provide most value to an agile project by looking for novel or unexpected risks. Such problems, when found, often reveal hidden complexities in the technologies used, or unintended consequences of design decisions. Testers need diagnostic and exploratory skills to chase down these important bugs.

Refactoring: *The system is re-structured, without changing its functionality, to improve simplicity or flexibility.*

Such changes should be invisible to testers confirming that the system works as before, and so do not impose maintenance costs. However, exploratory, bug-focussed testers will be interested in all such changes, as novel behaviours can result. For instance, if, after refactoring, an object behaves in functionally the same way, but in half the time, it may expose an otherwise hidden timing issue in another object.

Refactoring relies on comprehensive automated unit tests. Without such scaffolding, there can be no confidence that changes made to the code have indeed left the code working as before. Manual testing, whether by a skilled tester or not, is no substitute; it is not only prone to errors and omissions, it will not provide the instant feedback that the coder requires. Without refactoring, code can become bloated and inefficient, as the first, simplest solution becomes accreted with modifications and special cases. Code that cannot be refactored is immediately legacy code, and becomes a hindrance to the ongoing development of the product.

Continuous Integration: *The code is built into a working system very regularly, often daily, hourly or on each check-in. 'Breaking the build' with your newly checked-in code is a cardinal sin.*

For test specialists used to the death-by-a-thousand-cuts attrition of pervasively-buggy code, continuous integration seems a distant dream. When experienced, it is a revelation.

The latest code is always available. It becomes simpler to set up tests and to trigger events to observe their consequences. It is rare to find that swathes of functionality are unreachable behind a bug.

Unfortunately, the smoother the build and the cleaner the tests, the easier it is to assume that the deliverable also works perfectly. This confidence is misplaced. It is the job of the tester to give the project that confidence by doing a good job of looking for problems.

Avoiding errors

The following practices have a pervasive effect on the quality of the product by avoiding errors. The fewer errors, the fewer bugs. These practices are often the first to drop out of an agile project as it becomes clumsy.

Metaphor: *Development of the product is helped by a simple, shared set of stories that describe the system.*

Avoids interpersonal communication errors. Especially good for avoiding design problems and interfacing problems. For testers, these stories are of course a great starting point for tests, and can help to add weight to the consequences of a bug. However, a tester's idea of what a system *should* do is likely to go far beyond these simple stories.

Pair Programming: *Two people work on the same code, at the same time, at the same machine.*

Helps avoid coder mistakes, and by introducing instant reviews, tends to result in cleaner code with more comprehensive unit tests. It can be very useful to pair a test specialist with a coder.

40-hour Week: *Alert developers make fewer mistakes; regular late nights / long weeks are expected to introduce more trouble than they are worth.*

Good for avoiding design mistakes and avoiding introducing bugs as a side effect of fixes. The discipline of a regular week also serves to give perspective on the project; more time away seems to lead to better solutions to problems, and to a clearer vision for the end purpose.

Of all these practices, this is the one most frequently ignored. Some teams are enthused by concentrated late-night work, and some rigid 9-5 groups may actually be bored stupid. Look out for repeated long weeks.

It can be tempting, as a tester, to stay well after the coders have left. While this can be productive, it is important to remember that a purpose of testing on an agile project is to provide rapid feedback on the product. Staying on after the coders go home immediately introduces an overnight delay. If a tester in an agile team can't keep similar hours to the rest of the team, they should try to do the more collaborative parts of their job during core hours, when the whole team is available.

Coding Standard: *All the programmers need to write their code in the same way, using the same data description and code communication methods.*

Helps to avoid problems where different people use the same data for slightly different things, and also avoids problems where one person cannot understand another's code.

This practice seems, in contradiction of the preferences expressed in the agile manifesto, to prefer process and tools over individuals and interaction. That it is still included is testament to its power. The more people on an agile team, the more this standard matters to the overall quality of the product.

For the test specialist, this practice has great effects on code readability, and on test data.

Speed and communication

Agility is enabled by the following practices that lead to swiftness and accuracy by taking many small, confident steps. The following practices avoid both stagnation and leaps of faith.

The Planning Process/Game: *Swiftly and collectively arrive at at a sketch of what to do* ***next*** *by combining decisions about value from the customer and decisions about cost from the developers. The project is steered as knowledge arises, rather than pushed down a path decided when knowledge was scarce.*

Testers will not be able to develop detailed plans or automated tests more than one iteration in advance. As iterations are typically two weeks long, this can seem an impossible imposition to some testers. Automated unit tests and acceptance tests go some way to relieving the burden, but testers will be greatly helped by being able to:

- generate and load data as required,
- automate their tests to at least allow rapid execution of short chains of common actions
- use exploratory and diagnostic techniques to observe, trigger and isolate surprises

The testers are often the source of vital information that helps the customer to weigh up whether to commission work to address risks, or work to add capability. Unless testers have clear and regular communication with the customer, such information is unlikely to arrive as speedily as is needed, and poor decisions will result.

Small Releases: *Frequently deliver something that is of practical use to someone.*

Maintains momentum, and pleases someone every release. For testers, this practice tends to lead to small sets of focussed tests and a clear idea of whether something is working or not. At least initially, testing is a lightweight task.

However, if testers are expected (or expecting) to own the quality of a product, they may be expected (or expecting) to re-run all the tests for previous releases at every release. This ever-increasing requirement is compounded by the growing complexity and interaction of the system. This expectation should be modified – quality should be of core importance to everyone on the team, rather than the personal domain of a tester.

Wherever possible, regression tests and tests of fixes should be either made part of the automated test suite, or be understood to be one-off tests. The more tests go into the automated tests, the longer those tests will take, and testers will need to work with the rest of the agile team to develop a means of choosing those tests to be part of every build, and those to be run less frequently.

Simple Design: *Each iteration should deliver its functionality as simply as possible. Functionality to support possible later enhancements often avoided, and simplicity is maintained by "refactoring", discussed above.*

Functionality tends to be delivered as a piece, allowing testers to focus. However, see the note on refactoring above. As the simple design becomes more complex, or as a complex accretion of code is simplified, unexpected behaviours may be introduced that, by the nature of agile development, have not been picked up by the automated test suites.

Collective Ownership: *Anyone on the team can change any of the code.*

This includes the testers – and it may be that the testers will fix some of the bugs they find rather than log them for later attention. This saves time and effort, but may introduce long-term trouble if it leads to a skewed perspective of bugs, or to a test team that patches code rather than seeks out bugs.

On-site Customer: *A real user is part of the development team.*

This is a vital practice on an agile project, and improves speed and communication by providing fast answers to questions, immediate feedback on the product, and vital prioritisation/selection of tasks.

Test specialists will work closely with the customer, and may assist them in their test design and automation.

However, on projects with no customer, many customers, or an absent customer, the test specialists may take on the role of a customer, sometimes in conjunction with marketing staff, sales staff, business analysts or product champions.

Helping and hindering an agile effort

My experience of agile projects has been gained as a short-term consultant, as a long-term mentor, as an agile team participant with a specialisation in testing, and as a tester representing a customer.

However, many agile projects do not have anyone on the team with a background in testing. Testing is simply one of the roles that individuals take on in the course of their work. Bringing a skilled tester into an agile team can allow the whole team to understand the complexities of the role.

In this section, I look at the ways that testers can work within an agile team, then at the common problems that can face a tester on an agile team. I also list some of the ways in which even a well-meaning tester can damage an agile effort.

Being a tester

It can be strange to be a tester on an agile project; on many agile projects, no-one (or everyone) is a tester. There may not be a role for a test specialist, or you may find that everyone expects far too much.

If you've not worked on an agile project before, here are some of the experiences you may have:

- you will communicate more, with your coders and your customers.
- you will need to have (or will rapidly gain) a deeper understanding of the technology and the code
- you will no longer be the gatekeeper
- you will share more risk, and feel an increased sense of ownership
- you will personally fix more bugs

Where testers do have a role, it tends to fall into two parts;

- enhancing the confirmation tests at unit and acceptance levels
- identifying, diagnosing and exploiting unexpected behaviours

What sort of stories should a tester bring?

Testers will add to the stories that an agile team keeps to describe its goal product. Stories are a shared understanding, rather than a compete description, and you might find that your role is to add reasonable details to existing stories.

- User stories that involve edge cases
- User stories that involve malicious intent
- Stories that aren't for the users
- Infrastructural stories – ie loading the DB for setup
- Unexpected / unconsidered quality criteria
- Consequences after time / under load
- Experiments, rather than stories – like thought experiments?
- Bugs that need fixing, but can't be fixed right now

Enhancing unit tests

A tester on an agile project should be able to read the unit tests, and propose changes. Particular attention needs to be paid to situations where a unit test seems to be missing; a missing unit test may well mean that the corresponding code is also missing. This is particularly obvious – and particularly problematic – when code should be is in some way symmetrical, but is missing a test in one of its symmetric parts. Similar tests in other parts will pass, and the missing test will not trigger a failure.

Changes to unit tests might include picking values from a range of equivalents which stand a greater chance of failure; if a valid range is -10 to 10, existing unit tests might test at 10 and 11 – but it may be worth considering 0, -10 and -10.1 as subsidiary tests, as each may have a different reasons for failing validation.

Automating customer acceptance tests

As a tester, you may be able to bring automation skills to your customer, automating some or part of their acceptance testing. Apart from automating the actions of a user and programatically checking any output, you could introduce data load tools, combinatorial or pair-wise testing, and analysis and comparison tools across various outputs.

Being a Bad Customer

A bad customer does things that aren't 'happy path' interactions; they perturb and may break the system. It's woth considering whether a particular action is that of an incompetent

user, an expert/speedy user or a malicious user. A tester can use his or her knowledge of possible problems to be a bad customer. Some trigger ideas include:

- Empty sets, null inputs
- Invalid parts of input – characters, values, combinations
- Time changes
- Unusual uses
- Too much – long strings, large numbers, many instances
- Stop halfway / Jump in halfway
- Wrong assumptions
- Making lots of mistakes, compounding mistakes
- Using the same information for different entities
- Triggering error messages
- Going too fast

Role where there is no customer

A tester may play a substantial role as a customer advocate where:

- There is no customer – perhaps for a new product
- The customer is absent – if called away, on holiday or on business, or if the project has chosen (or has been forced) to have no customer
- There are many customers – many software products have a variety of users, and it may not be feasible to get a representative of each.

Exploratory Testing and Agility

Exploratory testing is risk-focussed, and is a good complement to the value focussed confirmation tests used in automated unit tests and acceptance testing. It is also a good fit with agile projects because it, too, suggests an alternative to relying on comprehensive documentation, and because both are based around concepts of learning. Agile processes and exploratory testing both allow context to drive decision-making, and both enable and rapid, sustainable and non-catastropic changes of direction. Both are seen as edgy, or innovative, both value invention of new process or adaptation of existing process over slavish following of standard process, and both need discipline and place the onus on the skill of the worker. Fast feedback in each enables iterative approaches – agility allowing iterations across code design,

ET allowing iterations across test design. Both can cover up for a broken project – but neither is a good fix.

Exploratory testing and diagnostic testing are important skills to bring to an agile team, improving the quality of bugs and their descriptions.

Common testing problems on an agile team

Agile projects have some specific problems for testers that go beyond a lack of comprehensive documentation, and the impracticality of planning much more than a couple of weeks ahead.

Decision fatigue

Practices such as *shared ownership* and *the planning game* are enabled by a stream of frequent, just-in-time decisions, and participants in agile projects are frequently considering the value of their current activity. This constant reflexive questioning – 'is this useful or not?' – can be fatiguing.

When decision fatigue sets is, such decisions will be avoided, taken carelessly, or not recognised as decisions at all. Some people are not happy making these decisions at any time, let alone frequently; decision fatigue will set in far sooner for these team members than for others.

Test specialists have their share of decisions to make, but will also be involved in revealing decisions to be made by the team as a whole. When fatigued, it can help to consider strategic goals – value and risk – to get out of the rut of trying to choose between mutually-unattractive available options.

Testing in the nth interation

The more is delivered, the more there is to go wrong in an unexpected way. Scripts ossify, and an adequate set of happy-path tests one iteration may be a poor selection in the next.

Fixes can introduce unexpected problems. Late fixes, fixes on fixes, and tactical fixes to strategic problems seem particularly prone to turn into unpleasant surprises. It is also common to see bugs return when a fix is backed out because it causes unintended problems that are equivalent to, or worse than the initial bug. Existing unit tests may give a false sense of security in unexamined areas.

After 3-6 iterations, there is too much to test from scratch, but the complexity of the system may well have reached a point where no part of the system is genuinely immune from new emergent bugs. It may be tempting to turn back to code freezes, waterfall approaches and testers as gatekeepers, but this is to acknowledge that the agile project has already become clumsy.

Testing within same iteration as coding

Agile projects need to have code tested in the same iteration as it is made; feedback on the code needs to be as fast as possible. However, when coding is squeezed for time, or when

working code is only available toward the end of the iteration, the team will be tempted to defer testing until the next iteration. The product is 'thrown over the wall' to a test team.

This not only fails to provide timely feedback, but by neatly separating the test specialists from the rest of the agile team, reduces communication within the team and damages the integrity of the team. By making the test specialists separate and a release behind, this 'silo' approach is closer to a stunted waterfall than to an agile approach. However, it is not uncommon in situations where the test skills are bought into the team late, or are outsourced to a consultancy, and can be the result of using over-specialised testers who are not able to contribute to an iteration until there is working code to test.

Automated unit tests and continuous integration will help the team to produce working code as soon as possible. If the test specialists are inactive at the start of the iteration, they may be underused rather than incapable; their skills can be usefully employed in pair programming concentrating on the integrity of the unit tests.

The goal of the team should be to produce great code, rather than the paired and dysfunctional goals of producing as much code as possible and finding as many bugs as possible. Measuring the test effort by the number of bugs found can be counter-productive; if the test specialists are involved early, bugs may be avoided.

When exploratory testing for risks reveals problems toward the end of an iteration, it is common to continue such investigation over into the next iteration. These investigations will be introduced as new stories to that next integration, and (if chosen for inclusion) should be done as early as possible to allow fast feedback to the coders, and to allow testing to move on smartly to the next task.

An agile project is a close team; everyone shares ownership of the code and of the quality of the product. If the testers are squeezed, it may be that they are becoming the gatekeepers of quality, and that the practice of collective ownership is compromised.

Conflicts in test management on an agile team

The responsibility for 'managing testing' is often delegated to one or two members of an agile team, rather than taken on by the team as a whole. The role is often aligned with the responsibility for non-unit-test tools, test environments and test data. People in this role will find themselves weighing conflicting choices. Their choices will be familiar from test effort on non-agile projects, but the short timescales of an agile project make the problems particularly acute.

Looking for new risks in the bulk of the system vs *testing the 'bleeding-edge' new parts of the system*. The team will need fast feedback on the new parts of the system, but should get plenty from their automated unit tests and customer acceptance tests. As the project becomes more complex and is re-factored, it is important that test specialists look for emergent behaviours introduced in the existing system by re-factoring or interactions with new functionality and data.

Finding novel problems vs *improving the diagnosis of existing problems*. A creative approach to testing may identify a diverse set of problems – but it may be more important to reproduce, diagnose, or exploit known, existing problems to help the team find the true source and impact.

Known problems may be avoided to allow testing to progress in novel directions, but note that a problem that is avoided for more than one or two iterations can become accepted by the customer on the project, even if it would not be accepted by the wider world. When a team becomes blind to a bug in this way, it may be a sign that the team has stopped learning.

Belief that passing tests = working code

A large, working set of unit tests can encourage the idea that the code works as designed. However, the code works as the *tests* are designed, but the design itself – or the tests as an expression of the design – may be flawed. If the sole assessment of an iteration is whether all the test passed, such flaws will not be visible.

The best way to write no bugs is to write no code. The best way to find no problems is to run no tests. Missing unit tests do not fail. Code reviews may identify code that has no tests, but a test review is needed to identify missing tests. Where coding is driven by writing the tests first (a common practice on agile projects), a missing test means missing code, but the customer acceptance tests may not be fine enough to recognise this, or may choose one route to functionality where the missing element is in another route to that functionality.

Mismatch between Agile Methods and Exploratory Testing

There is much common ground between exploratory testing and agile development, and the approaches are often used together. However, the match is not clean, and testers on agile teams should be aware of the following:

In an agile project, many tests confirm that the system works as expected. These tests are focussed on and driven by value. Exploratory testing is focussed on and driven by risk – unexpected behaviours. While the outcomes are complementary, the goals are conflicting, and this conflict should be recognised.

Agile testing tends to have all-automated tests with short feedback between test execution and test result, and short feedback between *code creation* and test result. The automation allows much longer feedback loops; automated unit tests are run frequently and should be able to be executed long after it has been designed, to check that no problems have been introduced. Exploratory testing has short feedback between *test creation* and test result. Tests are not designed to be repeated at a later date, and to allow the tester's brain to be fully engaged, are typically not fully automated, but are manual tests with automation support. This difference in approach can be difficult to explain to some agile teams – the value of test artefacts in an agile project persists over many executions, whereas exploratory tests are not repeated the same way twice..

Finding surprises by exploration can be hard in an agile project's short timescales from code creation to release. Although manual exploration may be a great way to observe unexpected behaviours, it is a slow way to trigger those behaviours. A tester on an agile team needs to be familiar with exploratory techniques and to be well-supported with tools.

Exploratory testers tend to be devoted, career testers. However, agile teams may have an antipathy to such a specialised beast, and a tester who can only test will need to be flexible and able to learn fast if he or she is to give value to an agile project without being able to interact directly with the code or design.

Right, but wrong

An agile effort can be interrupted by well-meaning, but inappropriate test work. It is worthwhile recognising this and finding alternate, less damaging approaches.

1) Logging every bug can be a desirable practice on many projects. On an agile project, it may be more efficient and direct to fix the bug – it is better to spend twenty minutes fixing and retesting a bug than to write up a bug report, triage it and turn it into a story, re-diagnose and re-find the bug before fixing it and retesting it. Paired testing with a coder can facilitate this, and some testers will need to relinquish their hold on bug metrics.

2) Bug metrics allow analysis of the problems found, but can be fatally skewed by bug avoidance and bugs fixed in unit testing. Better to keep track of the bugs that escape the team than the bugs that escape the coders.

3) Insisting on a code freeze is not going to work an an agile project. Instead, write flexible and maintainable tests, and learn to love reviewing unit tests.

4) In the same way, insisting on complete documentation is not going to work. Talk with your colleagues and interview your customer.

5) Independent testing will isolate the test effort and rapidly stymie an agile project with slow and incomplete feedback. Independent testing may have a use in the overall lifecycle of a product, but not within an agile development team. Communication and interdependence is the order of the day.

Learning and agility

The process of learning – about the product being built, about the technologies used for building it, about its value to the customer and the problems it may introduce – is of great importance to an agile project. This learning is based on feedback, and the information used in that feedback is often the result of testing.

To support their team, agile testers need to support and facilitate the learning process.

Can tools and procedures learn?

Tools and processes can smooth the path and enable fast feedback – but a rigid process or an inflexible tool will cause increasing friction in a changing environment.

Tools and procedures cannot learn, but can be changed. Lightweight shared procedures can be changed more easily than comprehensive, heavily documented procedures. Tools can be flexible only if the users of those tools are skilled in a range of their operations. It may be simplest to use a suite of tools that can be combined in a variety of different ways (ie tools within excel, or the standard suite of Unix tools) rather then becoming expert in a single swiss-army-knife tool.

Procedures and methodologies are frequently created or updated after problems, to avoid further, similar errors. They may be put in place to avoid a likely or potential error. As the agile effort matures, it will want to avoid more errors, and the procedures put in place may eventually choke it. It is important to consider re-factoring procedures and methodologies as the project progresses.

How do people learn?

People learn from feedback, from the consequences of their actions. Reflection is part of the process; effective learning is supported by prompt, clear feedback, both positive and negative.

Personal feedback can be encouraged by co-location and paired work. Procedural reviews of work products are a more heavyweight approach, and may not be so common on an agile project as direct, personal, immediate peer feedback. However, it is important to recognise and promptly stamp out hectoring or bullying. As a tester, it is important to criticise or praise the code, not the coder.

Real-time metrics – summary measurements that update as work is completed, not gathered at a later time for delayed perusal – provide instant feedback and so are an important component in enabling learning. They also are a powerful tool in allowing team members to understand the effects of their work on the overall project, and so give a sense of empowerment.

The product and the technologies used to build it will give feedback to an agile team member – test-first design is an iterative approach reliant on feedback from tests of a component of a system, and continuous integration introduces feedback from the technology of the overall product.

Can teams learn?

Teams learn as their constituent members learn, but as people enter and leave the agile team, the team may rapidly gain new skills, or find it has to re-learn old skills.

It is tempting to introduce procedures that codify the team's knowledge and so allow resilience to undesirable change – for instance, by beefing up the coding standard after every error found. However, if the rationale for new rules is lost and replaced by a dogmatic approach, resilience to undesirable change soon become resistance to all change, good or bad.

All teams develop folklore over time, memes and stories that act as shared lessons. These lessons slowly ossify from learned and shared experiences into received wisdom.

Procedures and folklore act to retain understanding by re-iterating it to old members and transmitting it to new members. To continue to learn, such understanding must be able to be challenged.

Enabling learning as a test specialist

Agile teams must take on the task of enabling learning as a team. However, test specialists produce information, and so have a special role to play in enabling learning.

- Give fast feedback on bugs, emergent behaviours etc – ideally just before or soon after the code is integrated with the main product. This may involve working closely with the coders as they complete their unit tests and code.

- Help coders extend their unit tests.

- Help the customer enhance their acceptance tests.

- Become involved with periodic project retrospectives to give feedback on the ways that different procedures helped or hindered the work from your point of view, and the overall effect on the product.

- Rather than promoting ever-increasing procedures to avoid errors, help those procedures to evolve through re-factoring

Further reading

Blogs:

Elisabeth Hendrickson:	http://testobsessed.com/
Brian Marick:	http://www.testing.com/agile/

Books:

Kent Beck:	Extreme Programming Explained: Embrace Change
James Shore:	The Art of Agile Development
Crispin, House:	Testing eXtreme Programming

Author's note

This paper is based on personal experience. I've worked with a number of teams that have consciously followed agile practices, and have gained my experience as a short-term consultant, as a long-term mentor, as an agile team participant with a specialisation in testing, and as a tester representing a customer. My first agile project was in 2002.

This is version 1 of this paper. In version 1.1, I hope to gain permission from some of those projects to use specific information and examples from those projects. Without the substance given by those real-world examples, this paper is simply an opinion piece. See my 2006 paper 'Things Testers Miss' for similar examples.

My thanks to all those I've worked with in agile teams for facilitating my own learning process. Thanks are also due to Anko Tijman and Evert-Jan Oosterhoff at Ordina for inviting me to deliver the talk that has become this paper.

Contact details

James Lyndsay, Workroom Productions Ltd.
Test Strategist

Email:	jdl@workroom-productions.com
AIM/Skype:	workroomprds
Mobile:	+44 7904 158 752
Landline:	+44 20 7871 9675
Fax:	+44 8717143877

Visit http://www.workroom-productions.com for papers, recorded talks and tools.