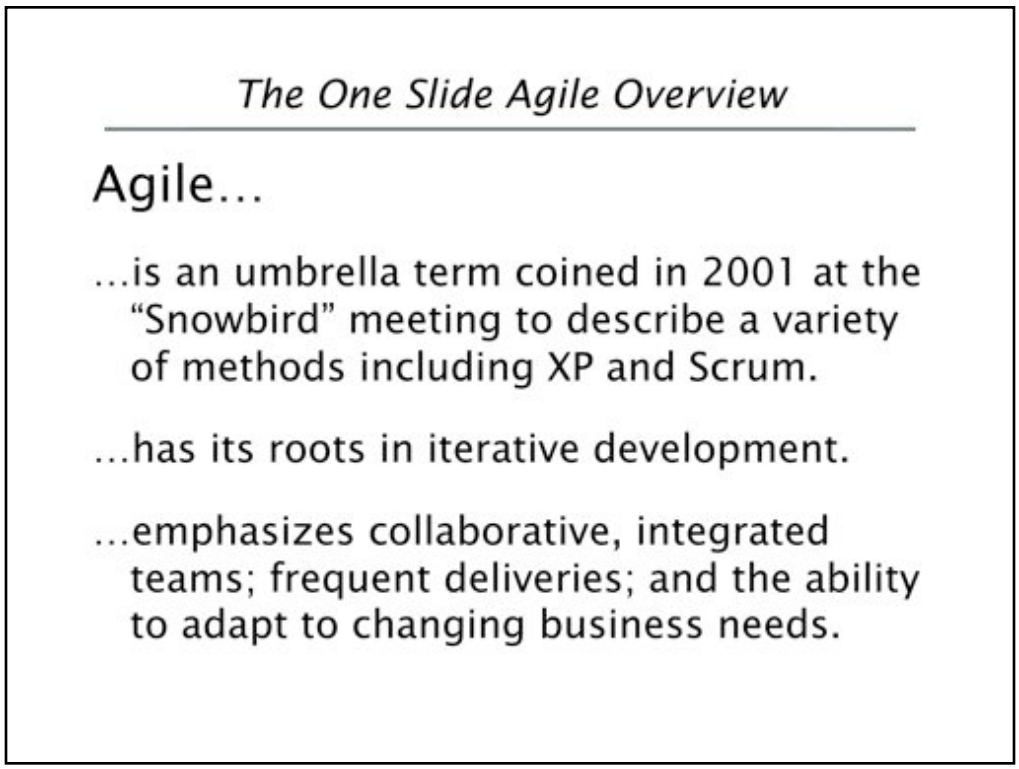
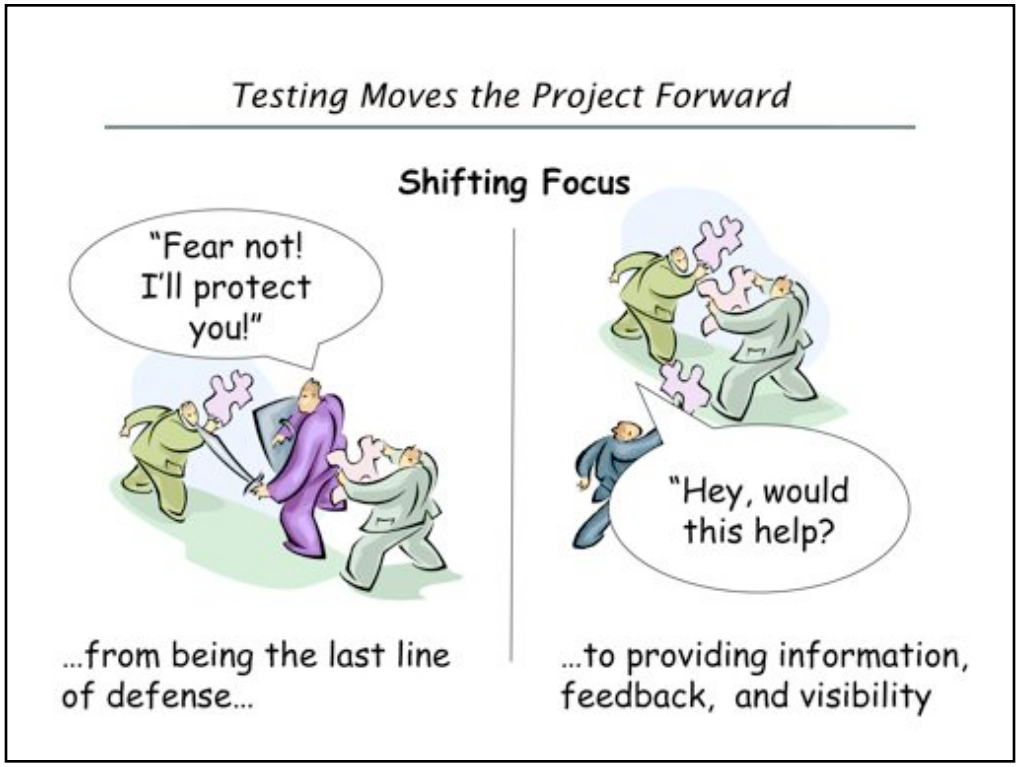


Agile Myths, Busted

Contrary to popular myth, Agile methods are not sloppy, ad hoc, do-whatever-feels-good processes. Quite the contrary. As Mary Poppendieck points out, speed requires discipline (see http://www.poppendieck.com/lean-six-sigma.htm). And Extreme Programming in particular is one of the most disciplined software development processes I've ever seen.

This means that some of the teams that claim to be doing "Agile" aren't. Compressing the schedule, throwing out the documentation, and coding up to the last minute is not Agile: it may result in short term speed but at the cost of long term pain. Agile methods are above all sustainable.

Agile teams really do need testers – or at least people who have strong testing skills. But there is a small grain of truth in the idea that Agile teams don't need QA. That's because Agile teams don't need is QA acting as a Quality Police. The business stakeholder – whether the Scrum Product Owner or the XP "Customer" – define what's acceptable and what's not. The QA or Test group supports the business stakeholder by helping them clarify acceptance criteria and understand risks.

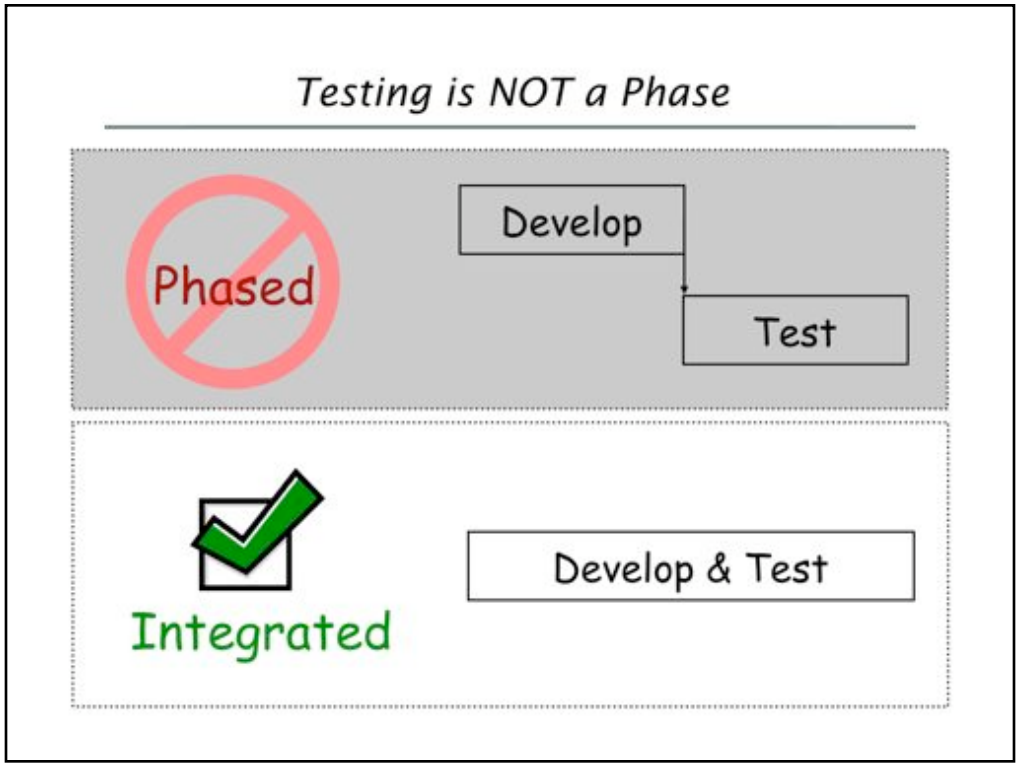


Testing Moves the Project Forward

On traditional projects, testing is usually treated as a quality gate, and the QA/Test group often serves as the quality gatekeeper. It's considered the responsibility of testing to prevent bad software from going out to the field. The result of this approach is long, drawn out bug scrub meetings in which we argue about the priority of the bugs found in test and whether or not they are sufficiently important and/or severe to delay a release.

On Agile teams, we build the product well from the beginning, using testing to provide feedback on an ongoing basis about how well the emerging product is meeting the business needs.

This sounds like a small shift, but it has profound implications. The adversarial relationship that some organizations foster between testers and developers must be replaced with a spirit of collaboration. It's a completely different mindset.

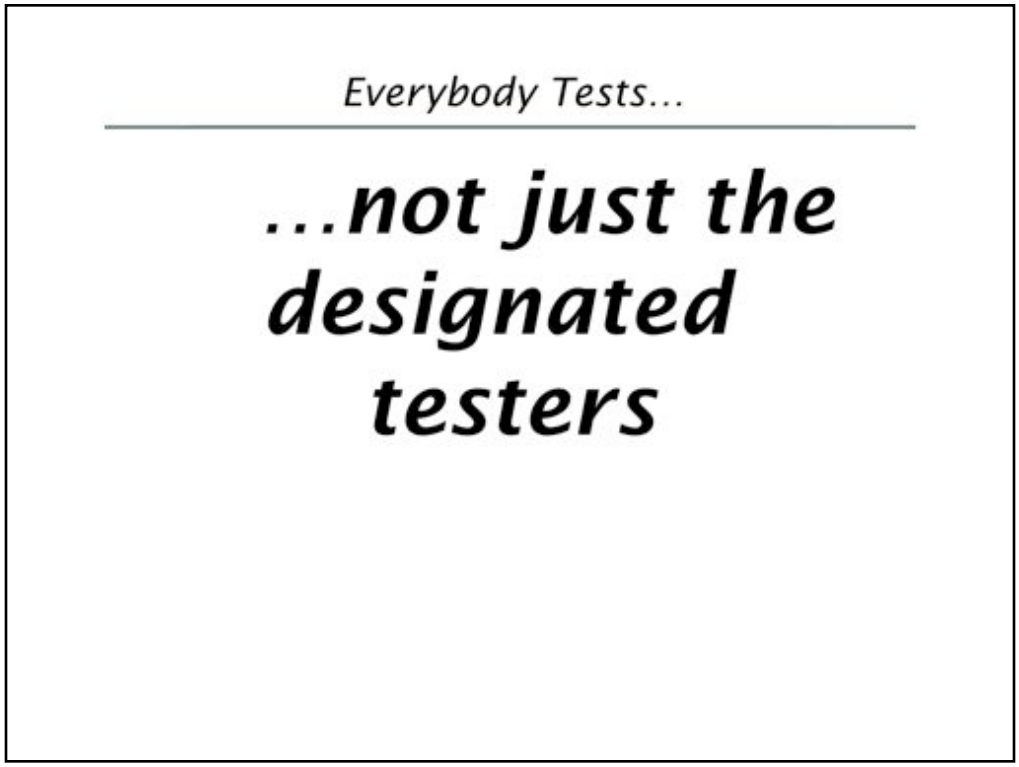


Testing is NOT a Phase…

…on Agile teams, testing is a way of life.

Agile teams test continuously. It's the only way to be sure that the features implemented during a given iteration or sprint are actually done.

Continuous testing is the only way to ensure continuous progress.

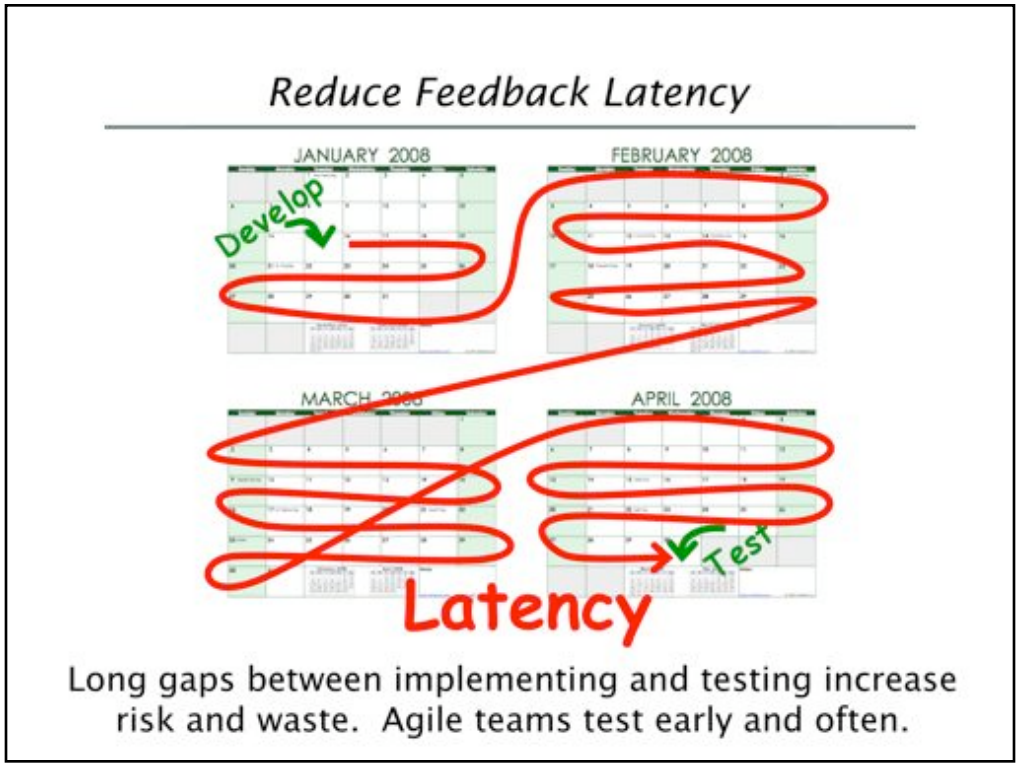


Everyone Tests

On traditional projects, the independent testers are responsible for all test activities. In Agile, getting the testing done is the responsibility of the whole team. Yes, testers execute tests. Developers do too.

The need to get all testing done in an iteration may mean that the team simply cannot do as much in each sprint as they originally thought. If this is the case, then Agile has made visible the impedance mismatch between test and dev that already existed. And that means that the team was not going as fast as they thought. They appeared to be going quickly because the developers were going fast. But if the testing isn't done, then the features aren't done, and the team just does not have the velocity they think.

Another way of thinking about this idea is that testing is the "herbie" on the team (see Goldratt's *The Goal*). Theory of Constraints says that the whole team can only go as fast as the slowest part. To go faster, the team has to widen the throughput of the slowest part of the process. Eliminate the bottleneck; everyone tests.



Shortening Feedback Loops

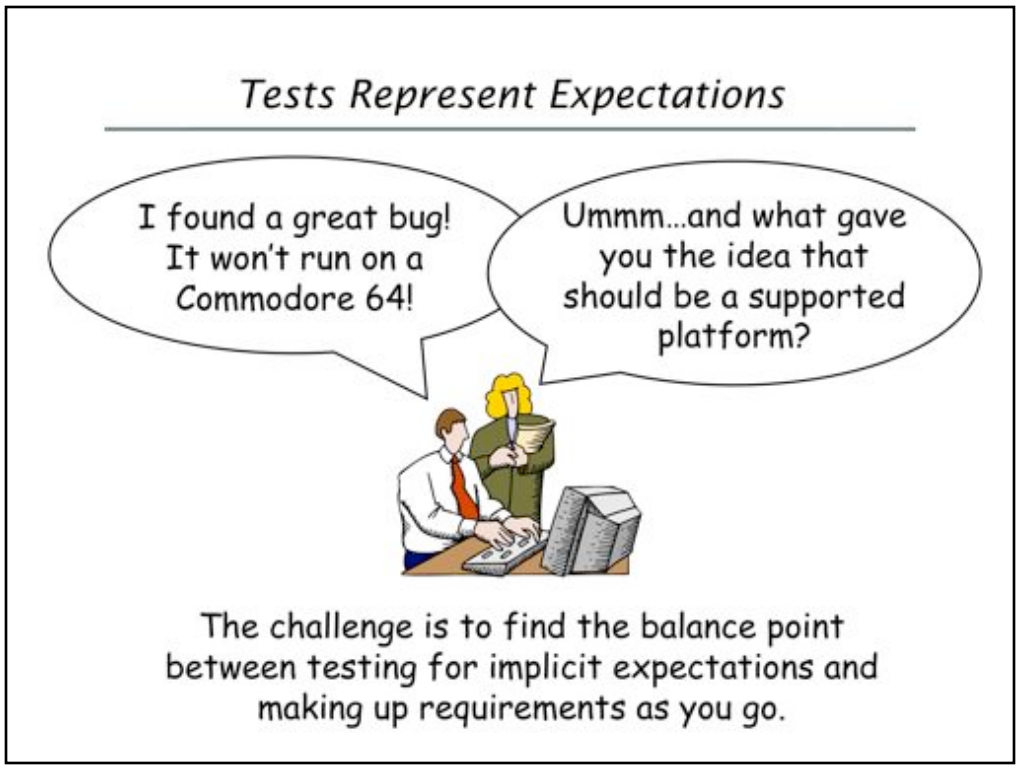
How long does the team have to wait for information about how the software is behaving? Measure the time between when a programmer writes a line of code and when someone or something executes that code and provides information about how it behaves. That's a feedback loop.

If the software isn't tested until the very end of a long release, the feedback loops will be extended and can be measured in months. That's too long.

Shorter feedback loops increase Agility. Fortunately, on Agile projects the software is ready to test almost from the beginning. And Agile teams typically employ several levels of testing to uncover different types of information.

Automated unit tests check the behavior of individual functions/methods and object interactions. They're run often, and provide feedback in minutes. Automated acceptance tests usually check the behavior of the system end-to-end. (Although, sometimes they bypass the GUI, checking the underlying business logic.) They're typically run on checked in code on an ongoing basis, providing feedback in an hour or so. Agile projects favor automated tests because of the rapid feedback they provide.

Manual regression tests take longer to execute and, because a human must be available, may not begin immediately. Feedback time increases to days or weeks. Manual testing, particularly manual exploratory testing, is still important. However, Agile teams typically find that the fast feedback afforded by automated regression is a key to detecting problems quickly, thus reducing risk and rework.



So Where Do Those Expectations Come From?

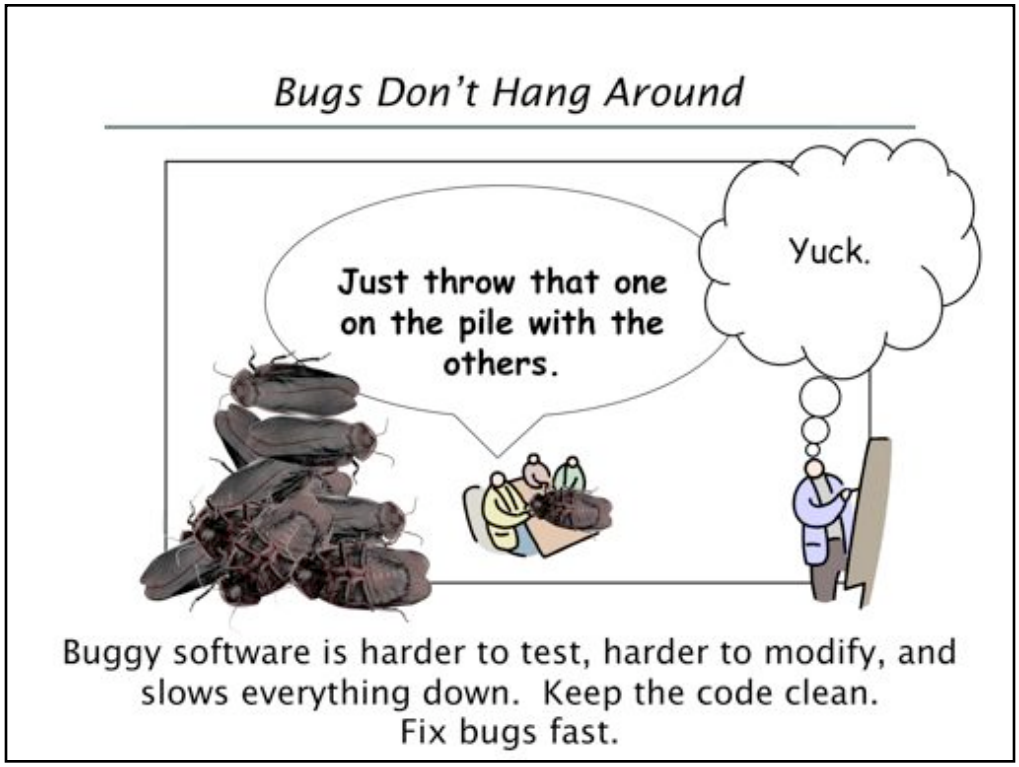
Once upon a time, before I started working on XP projects, I worked on a project where the developer protested "SCOPE CREEP!" to every bug report I filed.

Sadly, the two of us built up a lot of animosity arguing over whether or not the bugs I found were bugs or enhancements. I reasoned that I was testing conditions that were likely to occur in the real world, and "not crashing" did not count as an enhancement. The programmer argued that he'd done what he'd been asked to do and that it was too late to add more work to his plate. "No one said anything about the software being able to handle corrupt data!" he snapped.

I realized that the programmer thought I was making up new requirements as I went along.

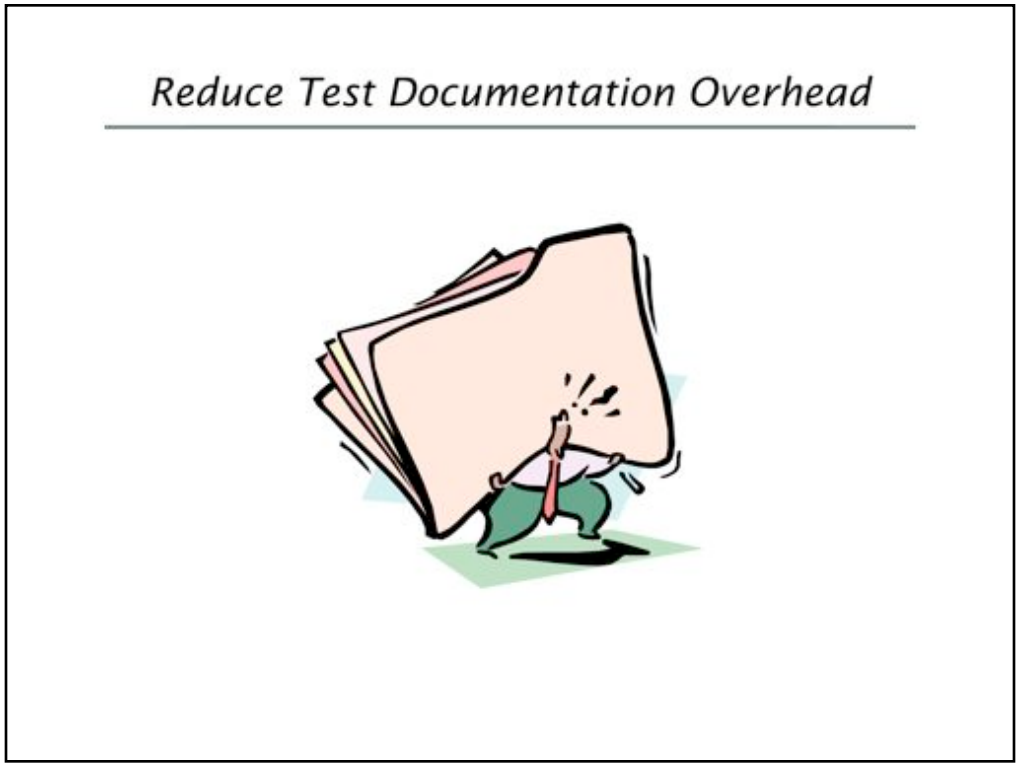
Of course, that's not what I intended. The way I saw it, my testing was revealing answers to questions no one had thought to ask before: What if this file is locked? What if that connection is broken? What if the data is corrupted? I would have asked the questions earlier if I could, but this was a waterfall-ish project, and testing happened at the very end of the process.

Working with XP teams has taught me that every test, whether manual or automated, scripted or exploratory, represents a bundle of expectations. Like the file tests I ran on that early project, sometimes those expectations represent implicit requirements (like "don't crash"). But sometimes my expectations turn out to be unreasonable. So now, before I spend a huge amount of time testing for a given type of risk, I ask questions to clarify my expectations with the project stakeholders.



Keep the Code Clean

This principle is an example of the discipline that Agile teams have. It takes tremendous internal discipline to fix bugs as they are found. If it's a genuine bug, as opposed to a new story, it is fixed within the iteration. To do otherwise is like cooking in a filthy kitchen: it takes longer to wade through the mess to do the cooking, and the resulting food may or may not be edible.



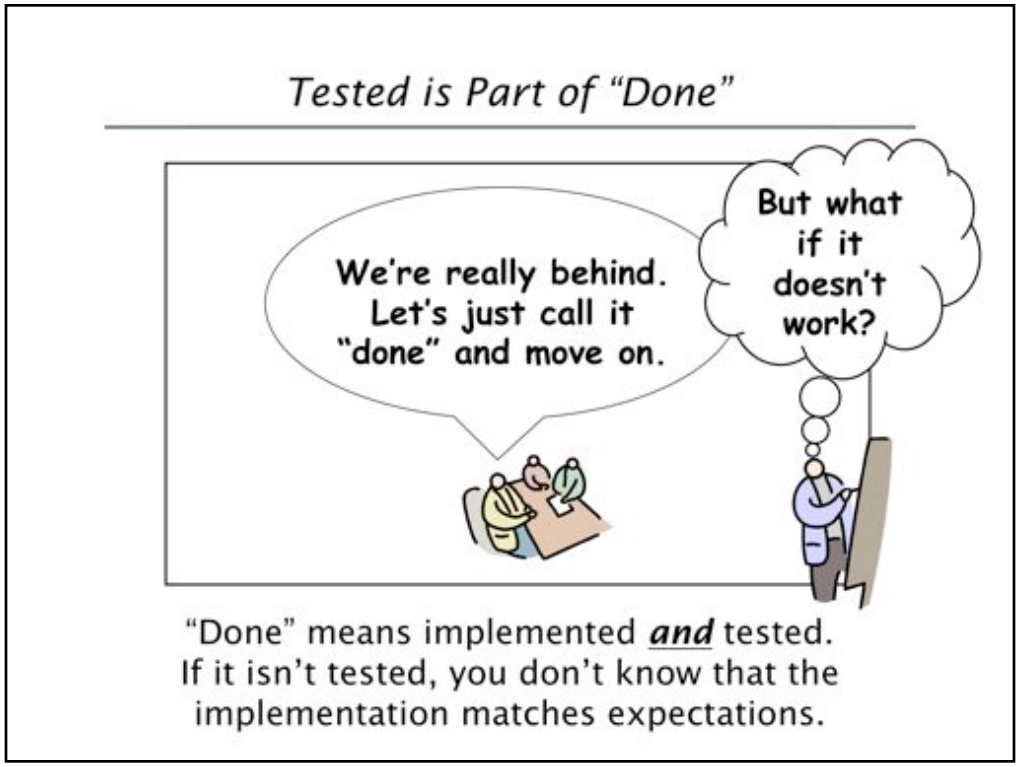
Lightweight Documentation

Instead of writing verbose, comprehensive test documentation, Agile testers:

- Use reusable checklists to suggest tests
- Focus on the essence of the test rather than the incidental details
- Use lightweight documentation styles/tools
- Capturing test ideas in charters for Exploratory Testing
- Leverage documents for multiple purpose

Leveraging One Test Artifact for Manual and Automated Tests

Rather than investing in extensive, heavyweight step-by-step manual test scripts in Word or a test management tool, we capture expectations in a format supported by automated test frameworks like FIT/Fitnesse. The test could be executed manually, but more importantly that same test artifact becomes an automated test when the programmers write a fixture to connect the test to the software under test.

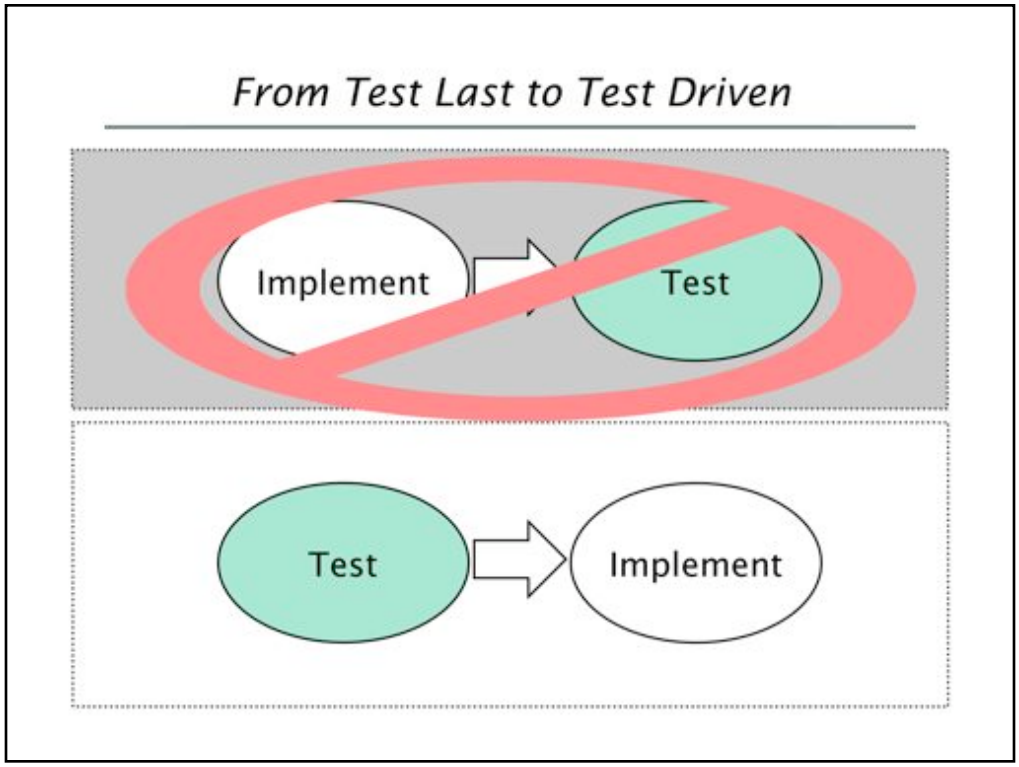


"Done Done," Not Just Done

In traditional environments that have a strict division between development and test, it is typical for the developers to say they are "done" with a feature when they have implemented it, but before it is tested.

Of course the feature isn't "done" until it's been tested and any bugs have been fixed. That's why there's a long standing joke in the industry that a given software release is usually "90% done" for 90% of the project. (Or, in other words, the last 10% of the effort takes 90% of the time.)

Agile teams don't count something as "done," and ready to be accepted by the Product Owner or Customer until it has been implemented ***and*** tested.

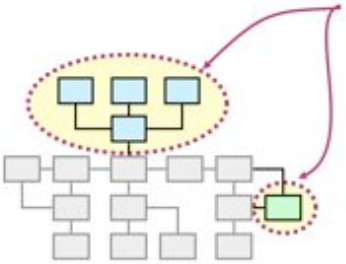


Test-Last v. Test-Driven

In traditional environments, tests are derived from project artifacts such as requirements documents. The requirements and design come first, and the tests follow. And executing those tests happens at the end of the project. This is a "test-last" approach.

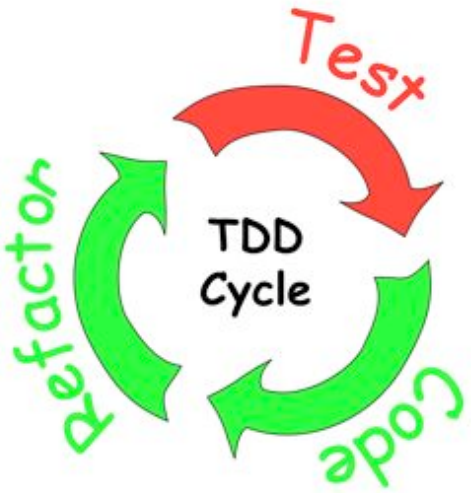
However, tests provide concrete examples of what it means for the emerging software to meet the requirements. Defining the tests with the requirements, rather than after, and using those tests to drive the development effort, gives us much more clear done criteria and shared focus on the goal. This test-first approach can be seen in the TDD and ATDD practices (see later slides).

Automated Unit/Integration Tests

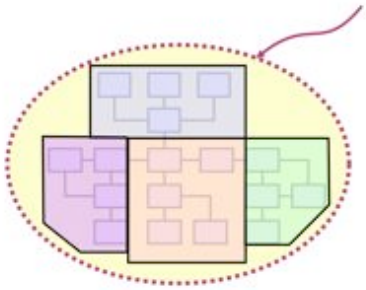


- Are ***code-facing***, written by programmers in support of the programming effort
- Are (usually) created using one of the xUnit frameworks
- Express expectations of the internal behavior of the code
- Isolate the element(s) under test
- Execute quickly
- Are executed often, with every change

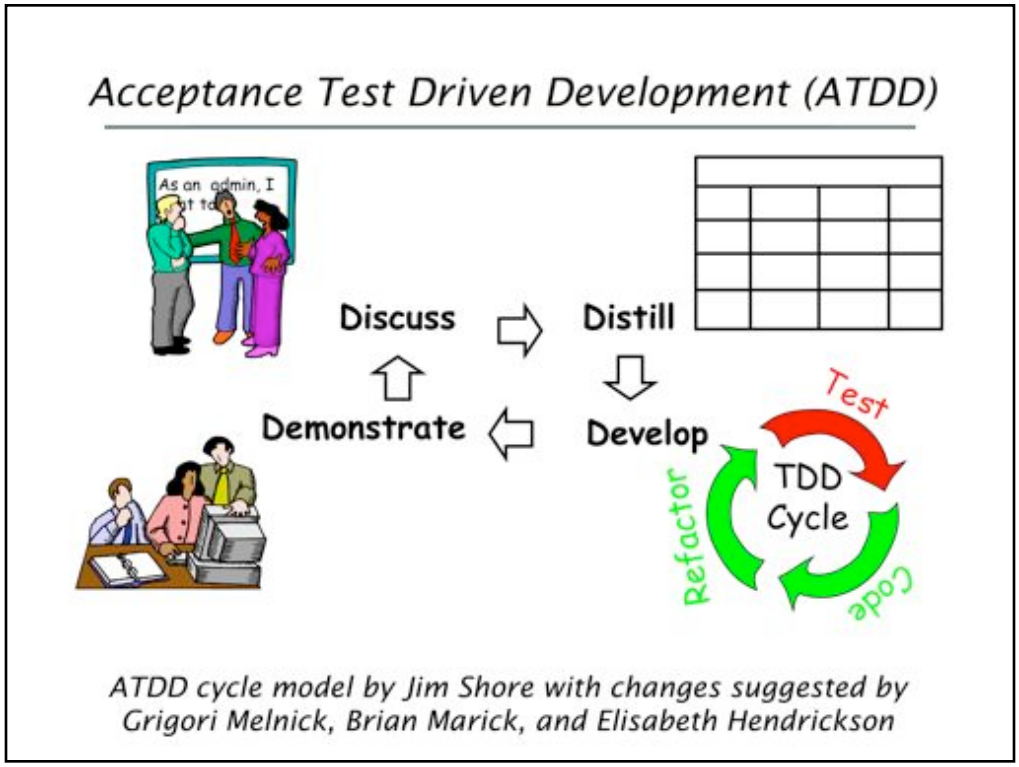
Test Driven Development (TDD)



Automated System-Level Regression Tests



- Are ***business-facing***, written by various members of the team in collaboration
- Express expectations about externally verifiable behavior
- Are (mostly) end-to-end
- Represent executable requirements
- Execute as part of the continuous integration process



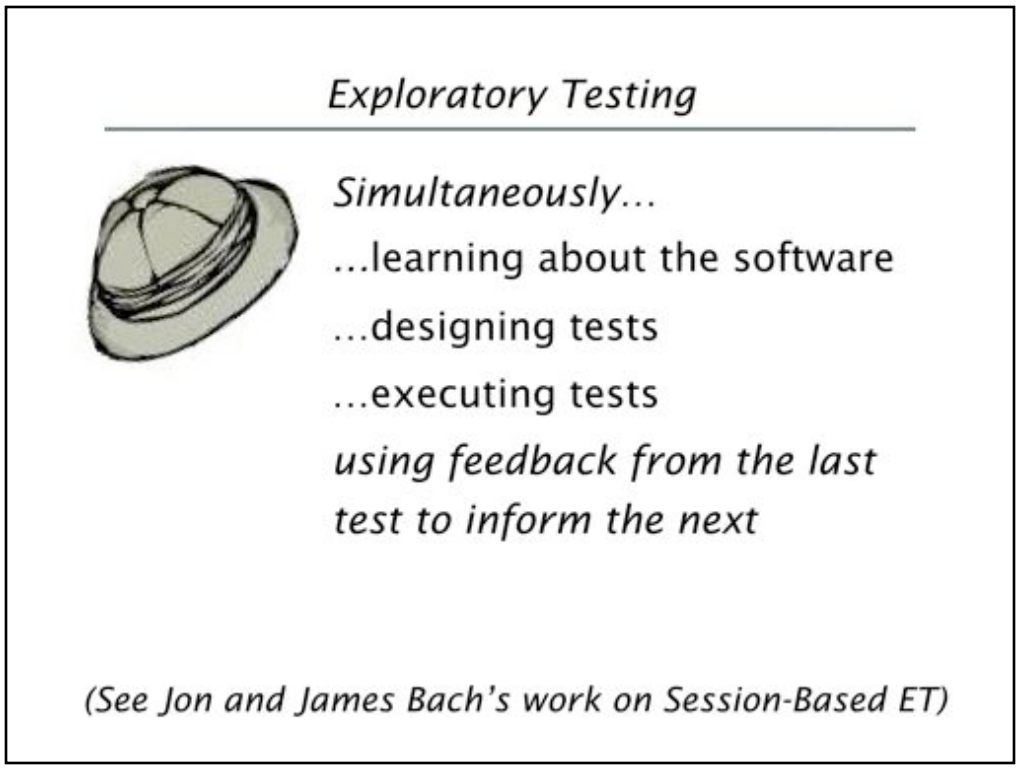
The ATDD Cycle

Discuss: work with the business stakeholders to understand their real needs and concerns. In traditional environments, this is usually called "requirements elicitation." In the context of Agile development, the purpose of this discussion is not to gather a huge list of requirements but rather to understand what the business stakeholder needs from one particular feature. During these discussions, ask questions designed to uncover assumptions, understand expectations around non-functional needs such as stability, reliability, security, etc., and explore the full scope of work the business stakeholder is requesting.

Distill: collaborate with the business stakeholders to distill their stated needs into a set of acceptance tests, or examples, that define "done." These tests should focus on externally detectable behavior and will be expressed in tables or keywords.

Develop: write the code to implement the requested feature using test-driven development (TDD).

Demonstrate: show the business stakeholder the new feature in the emerging system and request feedback.



A Short History of Exploratory Testing

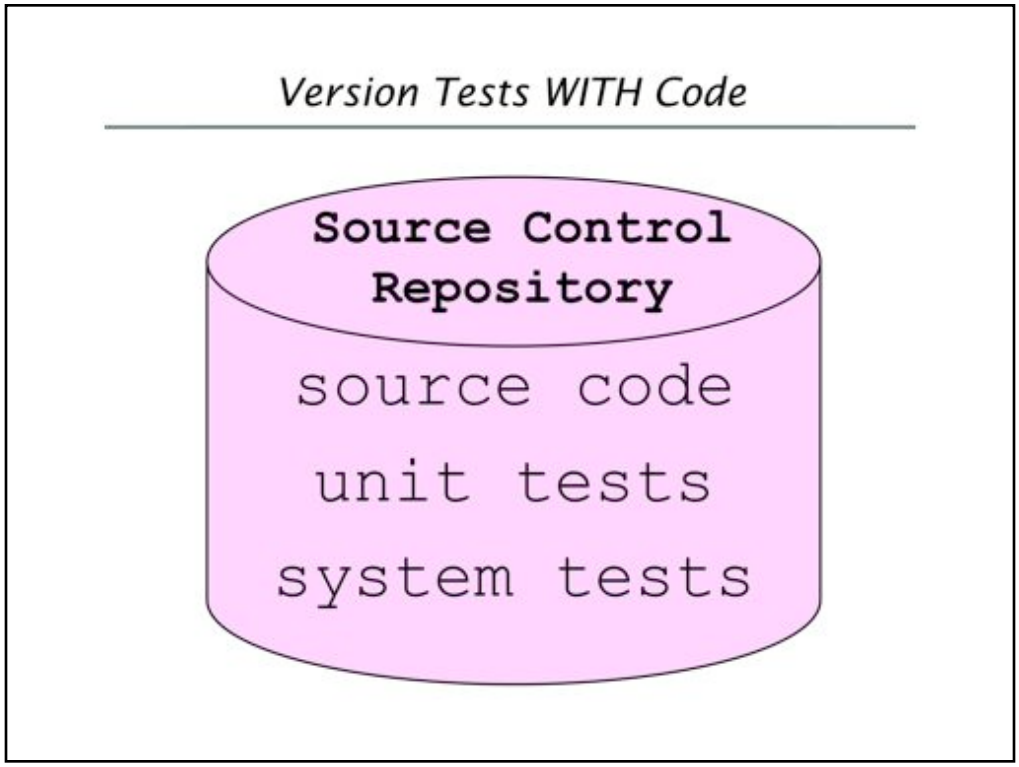
Cem Kaner coined the term "Exploratory Testing" in his book *Testing Computer Software*, although the practice of Exploratory Testing certainly predates the book.

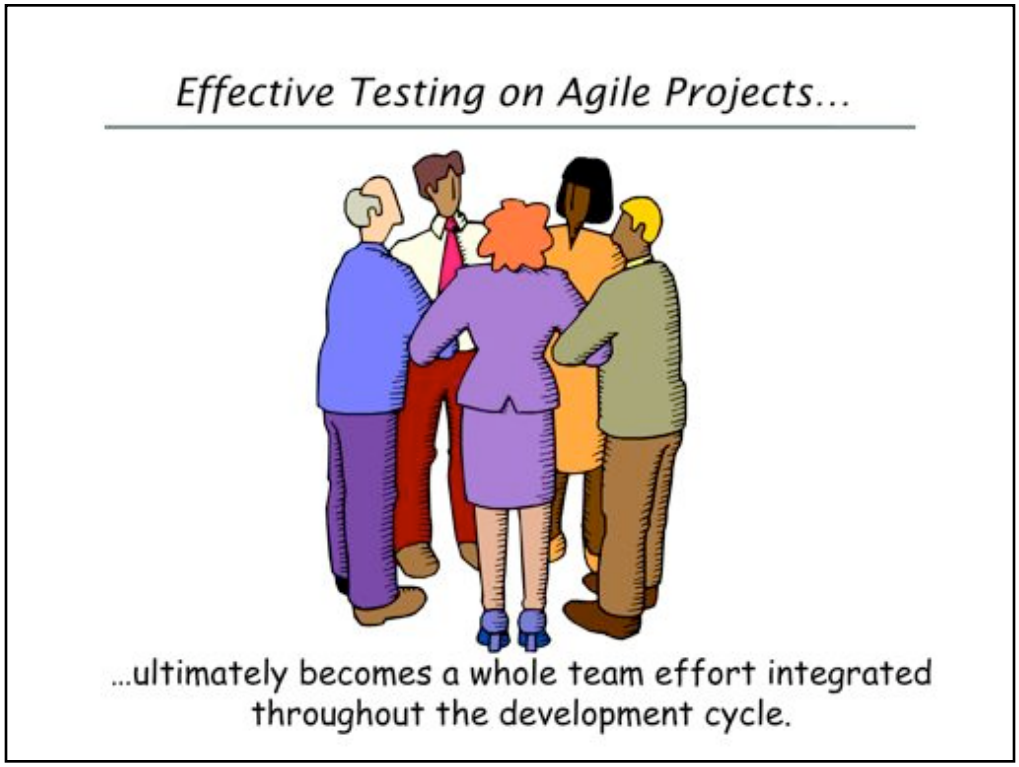
Since the book's publication two decades ago, Cem Kaner, James Bach, and a group of others (including Elisabeth Hendrickson and James Lyndsay) have worked to articulate just what Exploratory Testing is and how to do it.

Exploratory Testing Can Be Rigorous

Two key things distinguish good Exploratory Testing as a disciplined form of testing:

- Using a wide variety of analysis/testing techniques to target vulnerabilities from multiple perspectives.
- Using charters to focus effort on those vulnerabilities that are of most interest to stakeholders.





Collaborative Testing

Even before I started working with XP teams, I felt that it is important for testers to collaborate with all the other project stakeholders. In the course of my years in this industry, I have observed that isolation usually leads to duplicated and wasted effort.

Working on XP teams confirmed my beliefs. By integrating testing and development, we produced more solid code, more quickly, than I had seen on any of my past projects. Certainly there are contexts where independent testing is required, such as with safety-critical systems. But that doesn't mean the independent testers should be the only ones testing.

In XP, testing isn't a phase but rather a way of working so that at any given point in a project, you know that the work done to date meets the expectations stakeholders have of that work. And that requires a whole team effort.