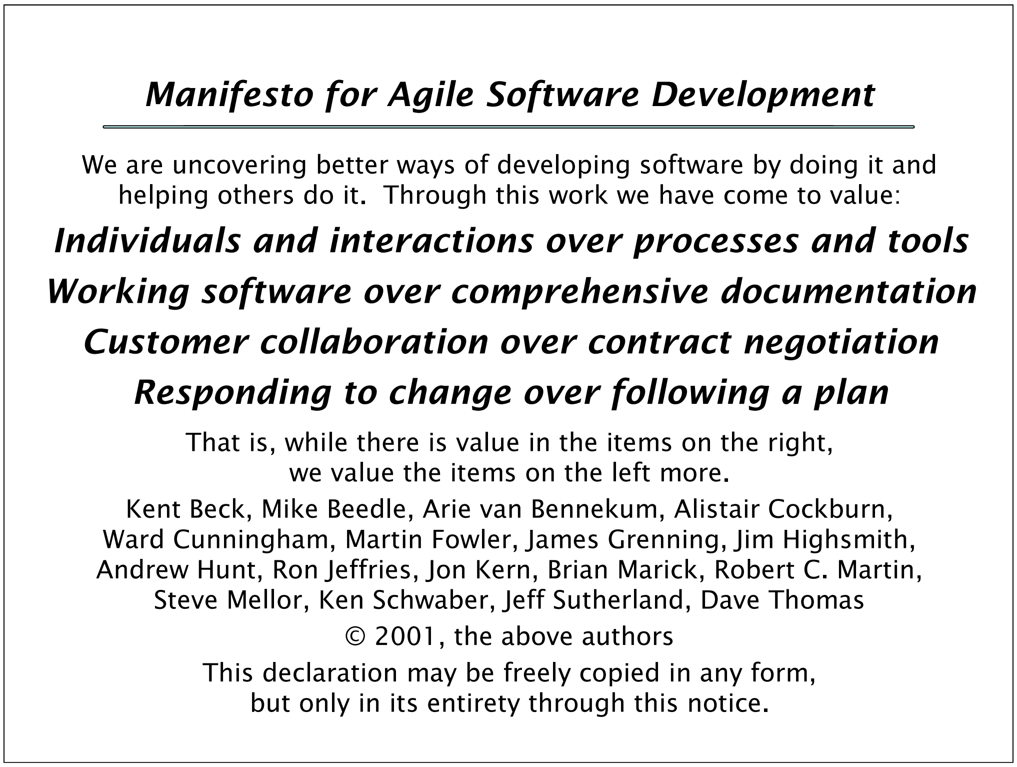
Agile QA/Testing

Elisabeth Hendrickson
Quality Tree Software, Inc.
www.qualitytree.com

Last updated November 20, 2006

Contents

- Agile overview
- How traditional testing practices evolved
- Agile and traditional projects contrasted
- Agile test practices evolving



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.

History of the Agile Manifesto

The signatories of the Agile Manifesto gathered at The Lodge at Snowbird ski resort in February 2001. Among the group were representatives from Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, and Pragmatic Programming.

The result was the Agile Manifesto, a set of statements that synthesize the core values of the various Agile methods.

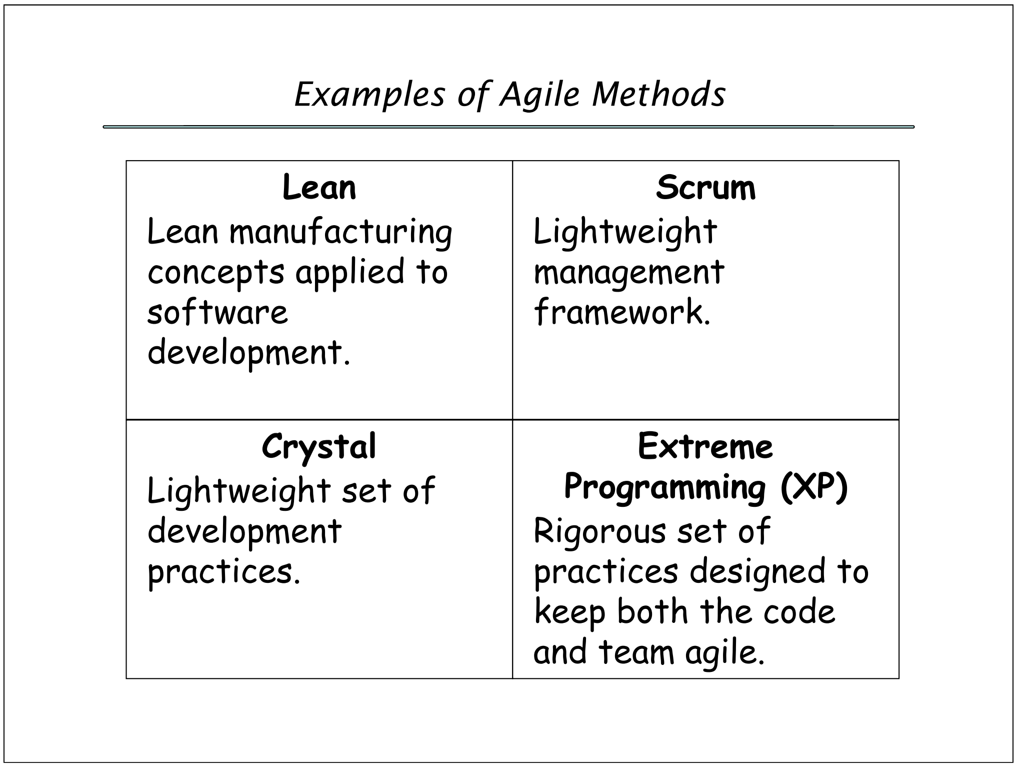
Agile Began Before 2001

Some people think that the Agile Manifesto kicked off the whole Agile Movement. This view is supported by the fact that from 2001 to today, there have been dozens of books published on Agile development under titles relating to XP, Scrum, Lean, Agile, and Adaptive.

However, although the creation of the Agile Manifesto may have been a turning point, it wasn't the beginning. The authors of the manifesto had been working in an Agile way for years before they articulated their shared vision for what Agile meant.

Even before that, in the 1980's Tom Gilb and Barry Boehm published works on Spiral and Evolutionary software development.

And in the 60's and 70's, software pioneers were Agile out of necessity. Gerald Weinberg notes that they used test-first development on the Mercury project, designing and implementing tests before code. For more information, see http://c2.com/cgi/wiki/wiki?HistoryOfIterative.



Examples of Agile Methods

Lean Lean manufacturing concepts applied to software development.	**Scrum** Lightweight management framework.
Crystal Lightweight set of development practices.	**Extreme Programming (XP)** Rigorous set of practices designed to keep both the code and team agile.

Agility and Agile Methodologies

"Agile" is a great buzzword. Who doesn't want to be agile? No one says, "thanks, I'd rather be inflexible and slow to respond." The problem with buzzwords is that people invoke them without understanding what the terms really mean. "Agile" is a relentless focus on providing a continuous stream of business value, even in the face of constant change.

For more details, see:

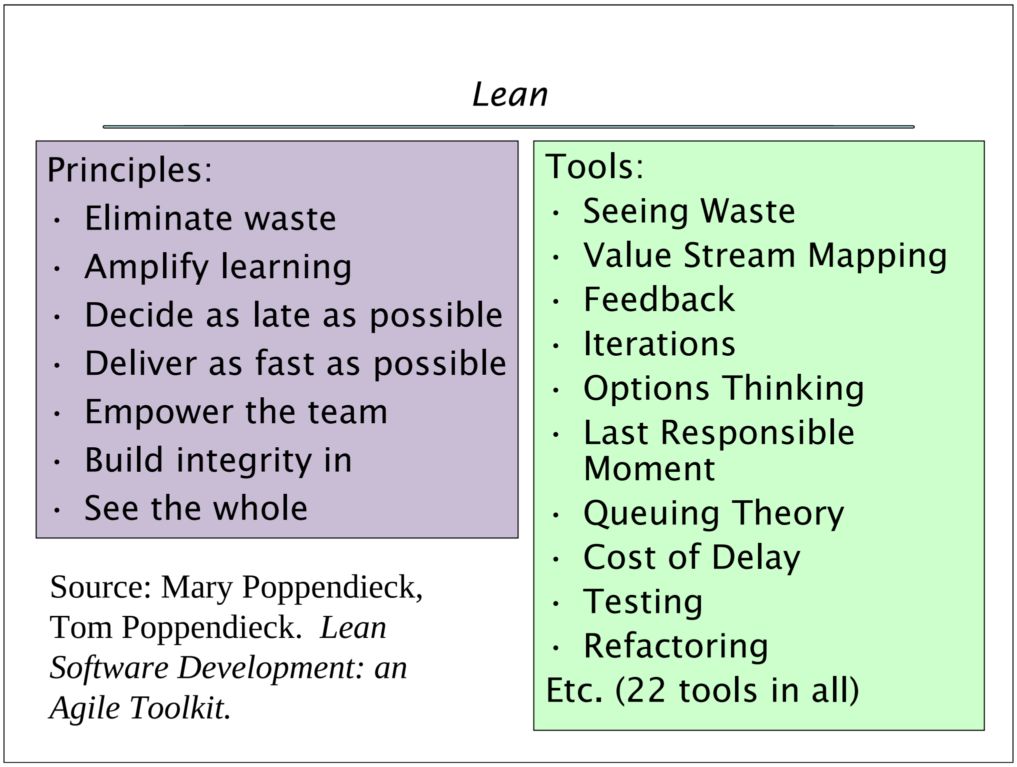
- The Agile Manifesto: http://www.agilemanifesto.org
- The Agile Alliance: http://www.agilealliance.org
- The Declaration of Interdependence: http://www.pmdoi.org/
- The Agile Project Leadership Network: http://www.apln.org

Agility is usually achieved by adopting one or more Agile methodologies.

Mixing and Matching Methodologies

Not all Agile methodologies address the same concerns. For example, Lean and Scrum are project-centric in that they provide managers with tools to monitor, prioritize, and control projects without specifying anything about how the software is built. By contrast, Crystal and XP are developer-centric in that they provide a set of good practices for a development team and less for a project manager.

Some organizations have tried combining methodologies with great success. One common combination is XP + Scrum.



Lean

Principles:

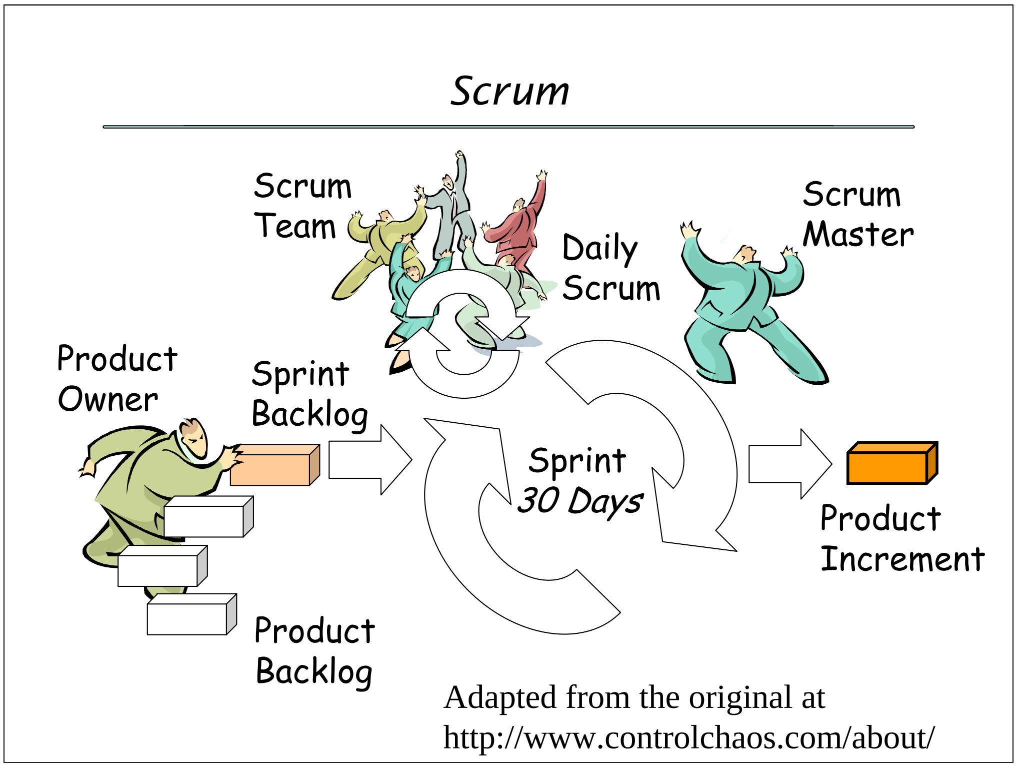
- Eliminate waste
- Amplify learning
- Decide as late as possible
- Deliver as fast as possible
- Empower the team
- Build integrity in
- See the whole

Source: Mary Poppendieck, Tom Poppendieck. *Lean Software Development: an Agile Toolkit.*

Tools:

- Seeing Waste
- Value Stream Mapping
- Feedback
- Iterations
- Options Thinking
- Last Responsible Moment
- Queuing Theory
- Cost of Delay
- Testing
- Refactoring

Etc. (22 tools in all)



Scrum

Scrum Team

Daily Scrum

Scrum Master

Product Owner

Sprint Backlog

Sprint 30 Days

Product Increment

Product Backlog

Adapted from the original at
http://www.controlchaos.com/about/

The Daily Scrum

Three questions each team member answers:

1. What have you done since the last Scrum?

2. What do you plan to do before the next Scrum?

3. What is getting in the way?

The Scrum Master Removes Obstacles

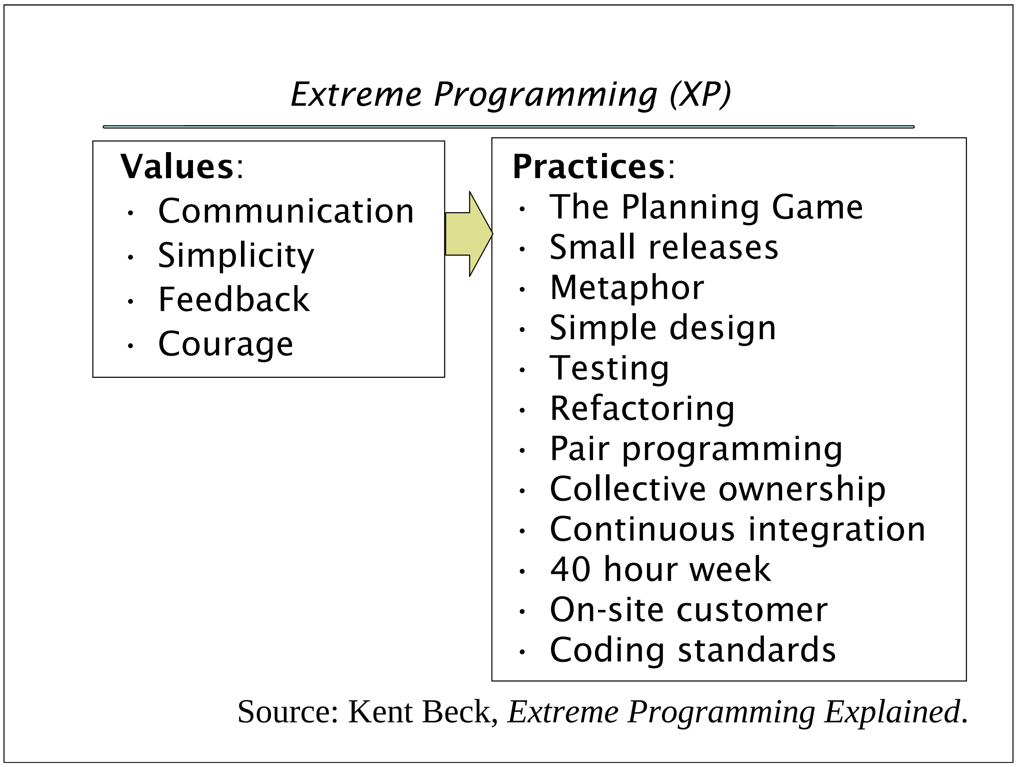
At one morning standup meeting, Elisabeth learned that one of the team members would not be able to deliver the code as promised. “What’s getting in the way?” Elisabeth asked. The team member reported that an executive had asked her to help resolve a tech support issue.

Resolving the tech support issue was important. But the executive who pulled the team member did so without taking into account the cost to the project. Elisabeth worked with the executive to find another resource who could help tech support, freeing the team member up to meet her commitments.

Crystal

> *The lead designer and two to seven other developers*
> *in a large room or adjacent rooms,*
> *using information radiators such as whiteboards and flipcharts,*
> *having easy access to expert users,*
> *distractions kept away,*
> *deliver running, tested, usable code to the users*
> *every month or two (quarterly at worst),*
> *reflecting and adjusting their working conventions periodically.*

Source: Alistair Cockburn, *Crystal Clear*. Available from http://alistair.cockburn.us/crystal/books/alistairsbooks.html

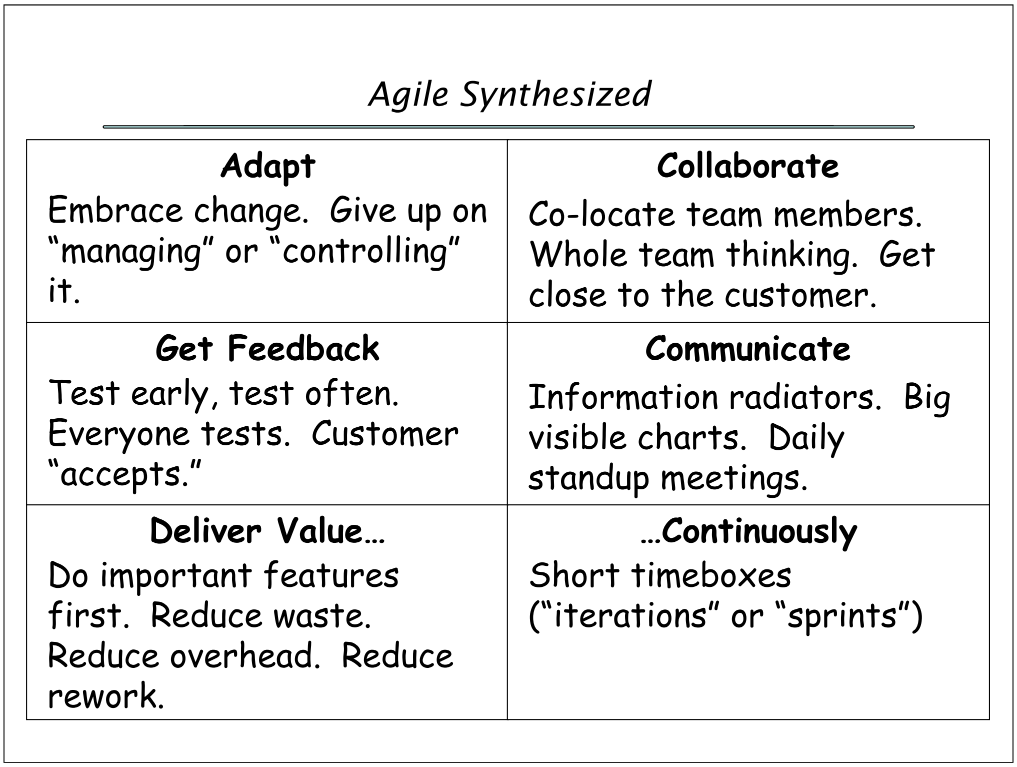


Extreme Programming (XP)

Values:
- Communication
- Simplicity
- Feedback
- Courage

Practices:
- The Planning Game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40 hour week
- On-site customer
- Coding standards

Source: Kent Beck, *Extreme Programming Explained*.



Agile Synthesized

Adapt Embrace change. Give up on "managing" or "controlling" it.	**Collaborate** Co-locate team members. Whole team thinking. Get close to the customer.
Get Feedback Test early, test often. Everyone tests. Customer "accepts."	**Communicate** Information radiators. Big visible charts. Daily standup meetings.
Deliver Value… Do important features first. Reduce waste. Reduce overhead. Reduce rework.	**…Continuously** Short timeboxes ("iterations" or "sprints")

Agile Expectations

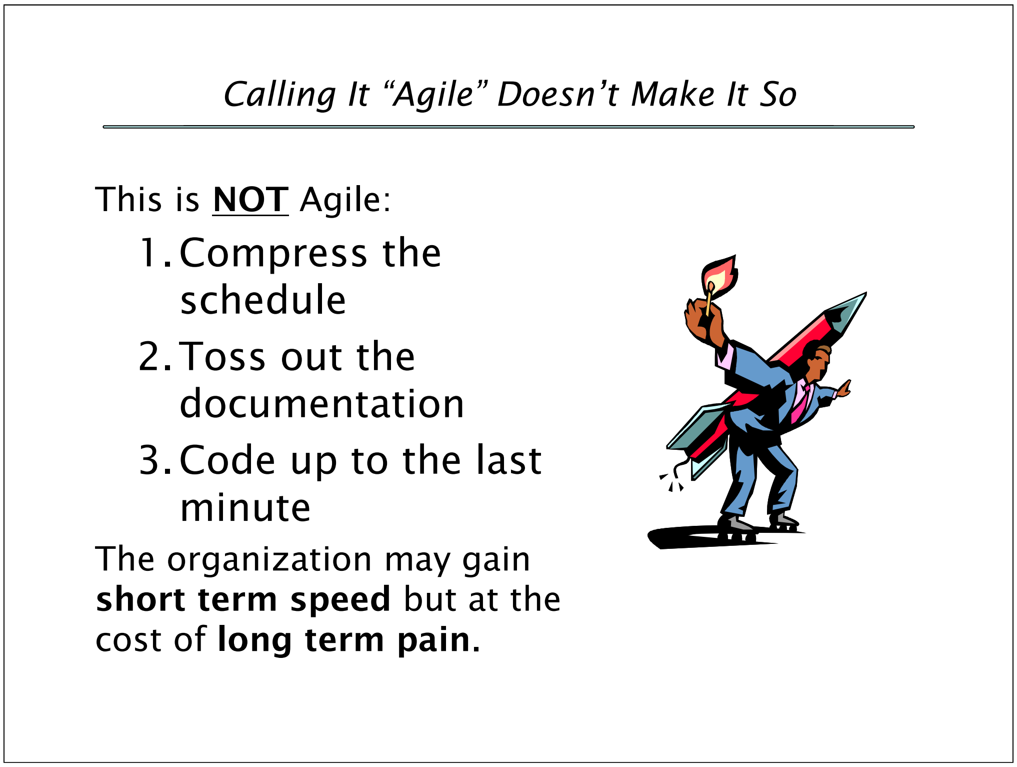
Change is inevitable.

The whole team, including the programmers, testers, customers, is responsible for the outcome.

Everyone on the team is accessible and actively communicating throughout the project.

Programmers test early, often, and aggressively.

The whole team actively solicits feedback.



Calling It "Agile" Doesn't Make It So

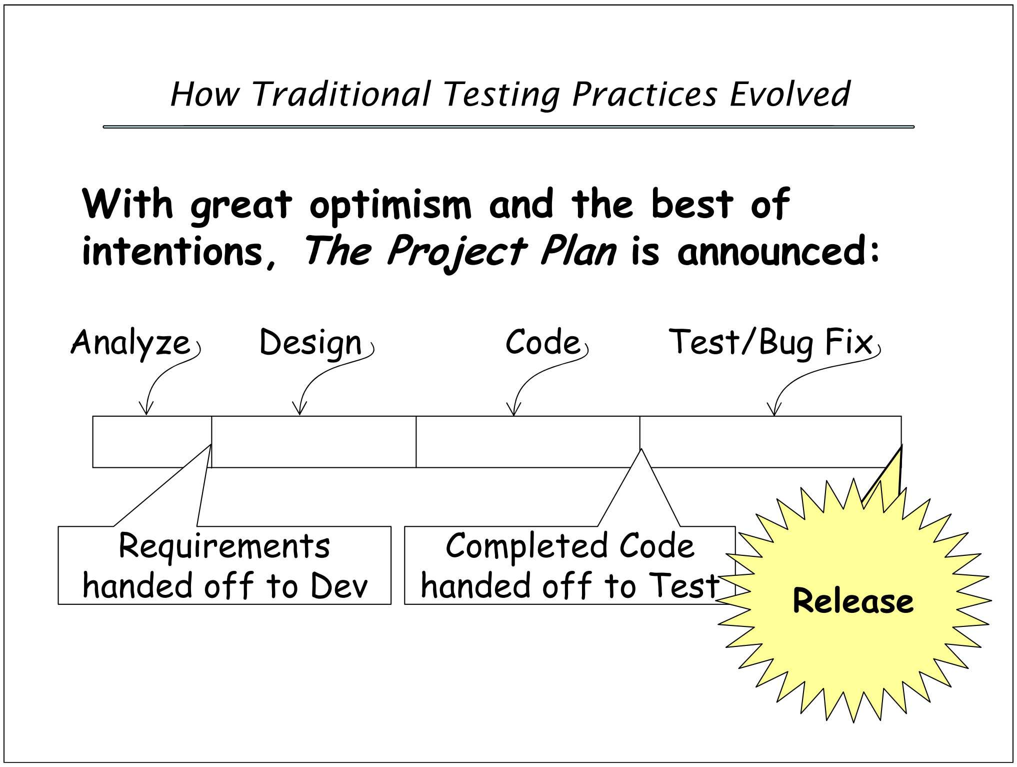
This is **NOT** Agile:

1. Compress the schedule
2. Toss out the documentation
3. Code up to the last minute

The organization may gain **short term speed** but at the cost of **long term pain.**

Agile Lip Service

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." Calling it Agile doesn't make it Agile.



How Traditional Testing Practices Evolved

With great optimism and the best of intentions, *The Project Plan* is announced:

Analyze

Design

Code

Test/Bug Fix

Requirements handed off to Dev

Completed Code handed off to Test

Release

Phased Development in an Ideal World

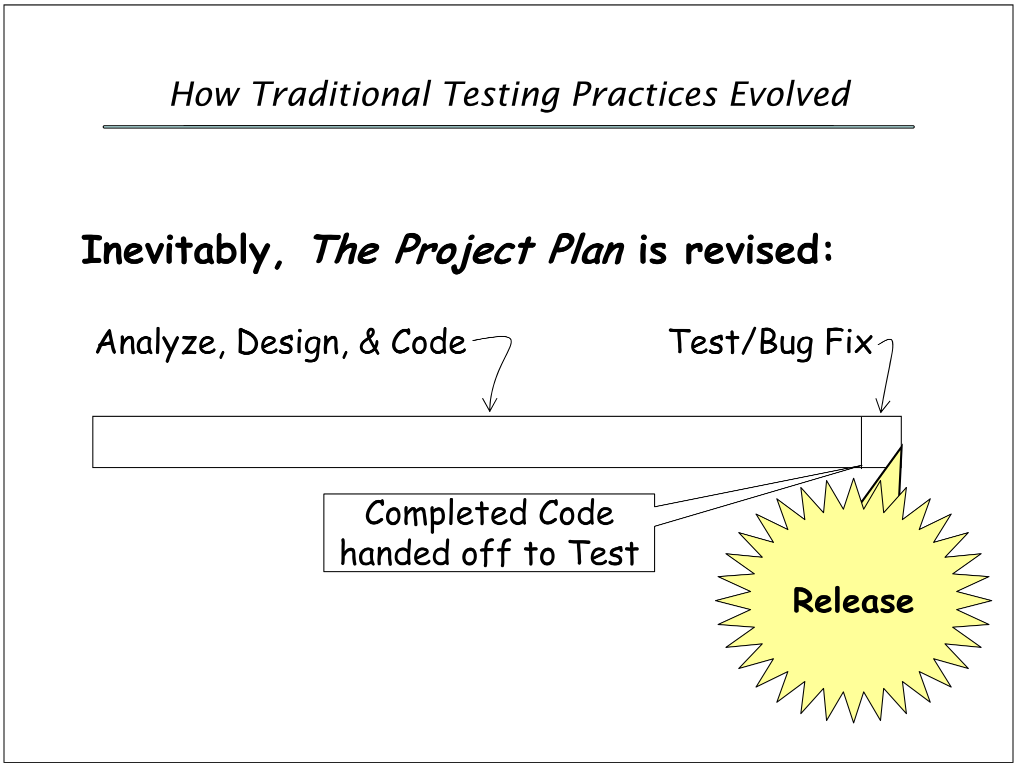
If we lived in a perfect world, it would be possible to know all the requirements up front. We'd get them right every time. We'd somehow uncover all the implicit assumptions and ambiguities. We'd have the power of perfect prediction to know every permutation of configurations and sequences that will occur in production.

In a perfect world, our designs would be perfect the first time. We'd know exactly where the performance bottlenecks were going to be, and would create a perfect architecture to compensate accordingly.

In a perfect world, our code would always do what we wanted it to, the first time. We'd have perfect understanding of all the 3rd party libraries we rely on and would use them correctly, and make the calls in the right sequence.

In a perfect world, whatever small issues remained at the end of the cycle would be easy to find and fix in plenty of time before the release. The fixes would be minor, requiring no major refactoring, no huge architectural changes.

Alas, we don't live in a perfect world. Technology, being made by humans, is especially imperfect Inevitably, unexpected discoveries lead to rework and schedule slips. Testers have learned that no matter what the project plan says, the reality is likely to be…



How Traditional Testing Practices Evolved

Inevitably, *The Project Plan* is revised:

Analyze, Design, & Code

Test/Bug Fix

Completed Code handed off to Test

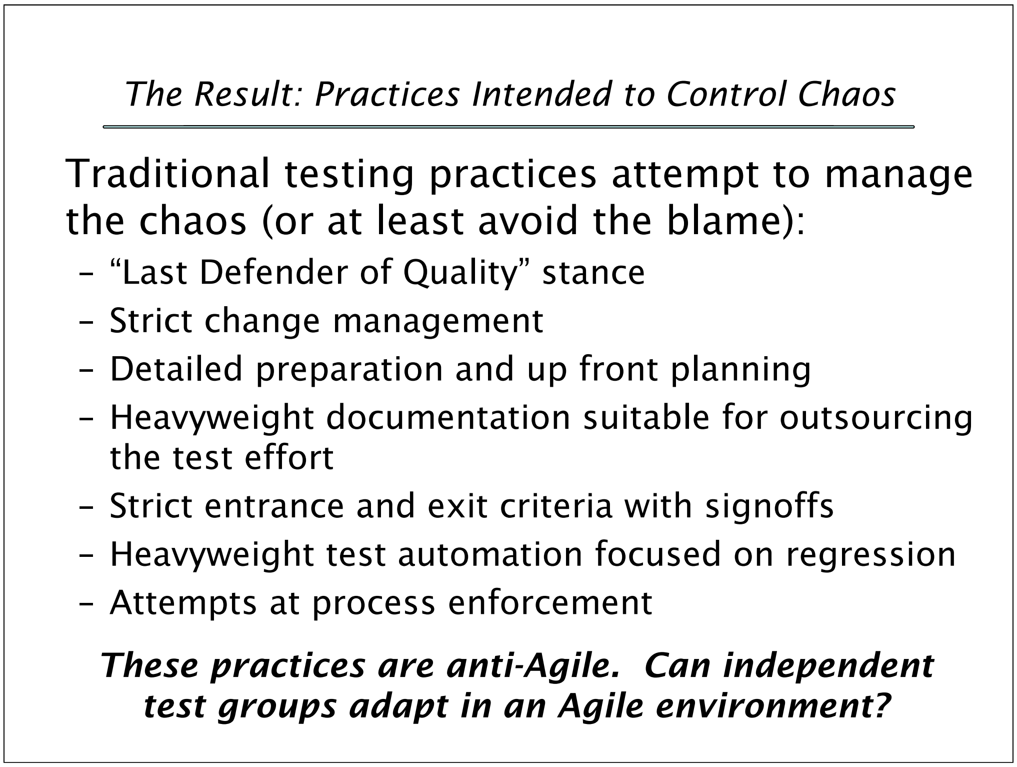
Release

The Problem of Late Testing

When the bulk of the test effort is squeezed into the last few months or weeks of a long release cycle, the time for testing evaporates with each successive schedule slip in preceding phases. The testers know that the less time there is to test, the less they will be able to test. And the less testing they can accomplish, the less the organization will know about the software before it's released. Less information means more risk.

For years, testing organizations have sought ways to do more testing in less time. Common approaches include: test automation, outsourcing, insourcing, bug bashes, and internal or customer beta tests to augment the internal test effort. Let's be honest: while some organizations have seen some gains from these practices, most organizations are still overwhelmed with too much to test and too little time in which to do the testing. Even if the testers are able to complete the testing, there often is no time left to fix the bugs testing reveals.

Although these problems are widely recognized, real solutions have proven elusive. Yet we've developed a number of practices aimed at controlling the end-of-release chaos.



The Result: Practices Intended to Control Chaos

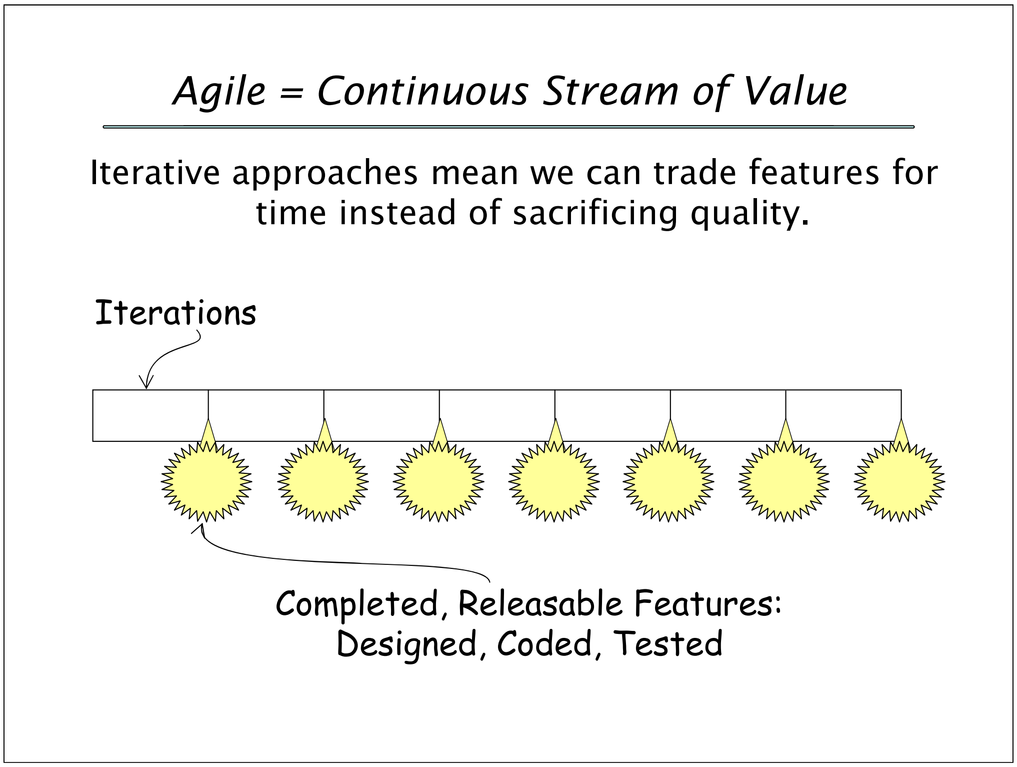
Traditional testing practices attempt to manage the chaos (or at least avoid the blame):

- "Last Defender of Quality" stance
- Strict change management
- Detailed preparation and up front planning
- Heavyweight documentation suitable for outsourcing the test effort
- Strict entrance and exit criteria with signoffs
- Heavyweight test automation focused on regression
- Attempts at process enforcement

These practices are anti-Agile. Can independent test groups adapt in an Agile environment?

The Solution to the Late-Testing-Problem

It's a deceptively simple solution. If delaying testing until the end of the release is risky…don't delay the testing until the end of the release. Easy to say, hard to do—unless you're in an Agile environment that supports early testing. Even then, it can be difficult to get testers involved from the beginning.



Agile = Continuous Stream of Value

Iterative approaches mean we can trade features for time instead of sacrificing quality.

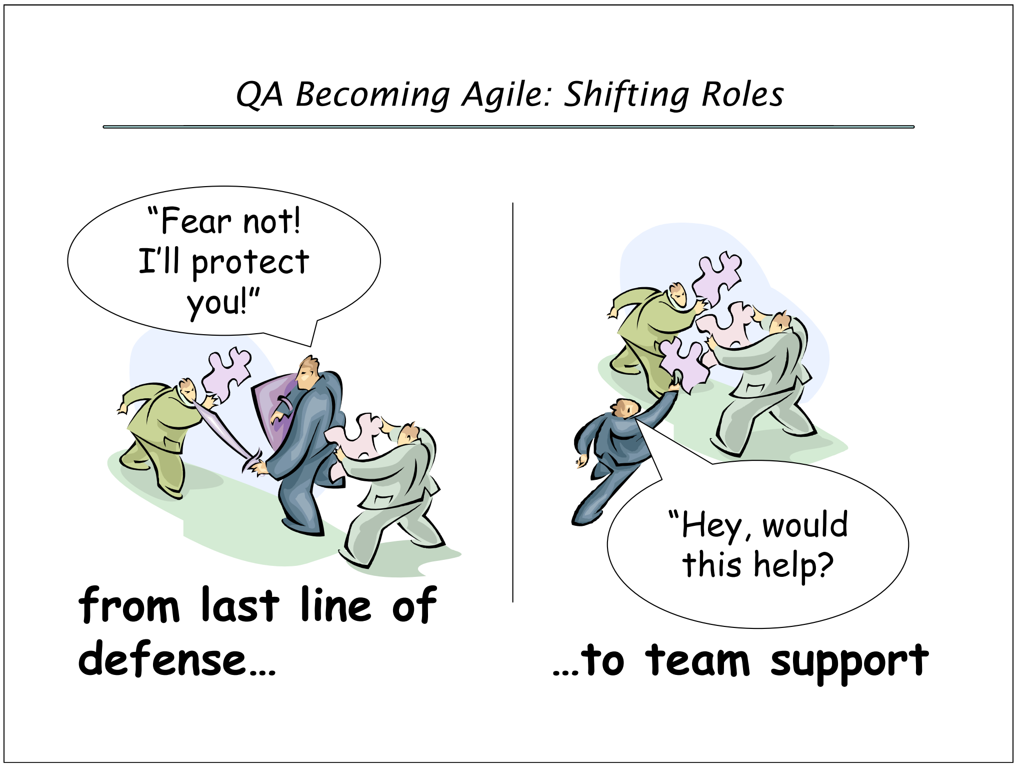
Iterations

Completed, Releasable Features:
Designed, Coded, Tested

Necessary Conditions for Adopting Agile

Here's how to tell if an Agile approach would work in your environment:

- The organization is willing to embrace agility as defined by the Agile Manifesto. Saying "Be More Agile" or "Go Faster" isn't enough. The team must be more interested in usable feedback than comprehensive documentation or detailed plans, and more interested in collaboration than signoffs.
- Everyone tests, not just designated testers. Agile teams are "test infected." (An important side effect of teams being test infected: the software is built to be testable from the beginning.)
- The whole team is responsible for quality, not just the testers or people with "Quality" in their title. Which are you more likely to hear: "How did you miss that bug?!?" or "What can we do next time to prevent something like that happening again, or detect it earlier?" When everyone is responsible for the outcome, there's more collaboration and less finger-pointing.
- Managers focus on fixing problems, not blame. Agile practices don't provide CYA paper trails and are unlikely to succeed in a high-blame, high-fear environment.



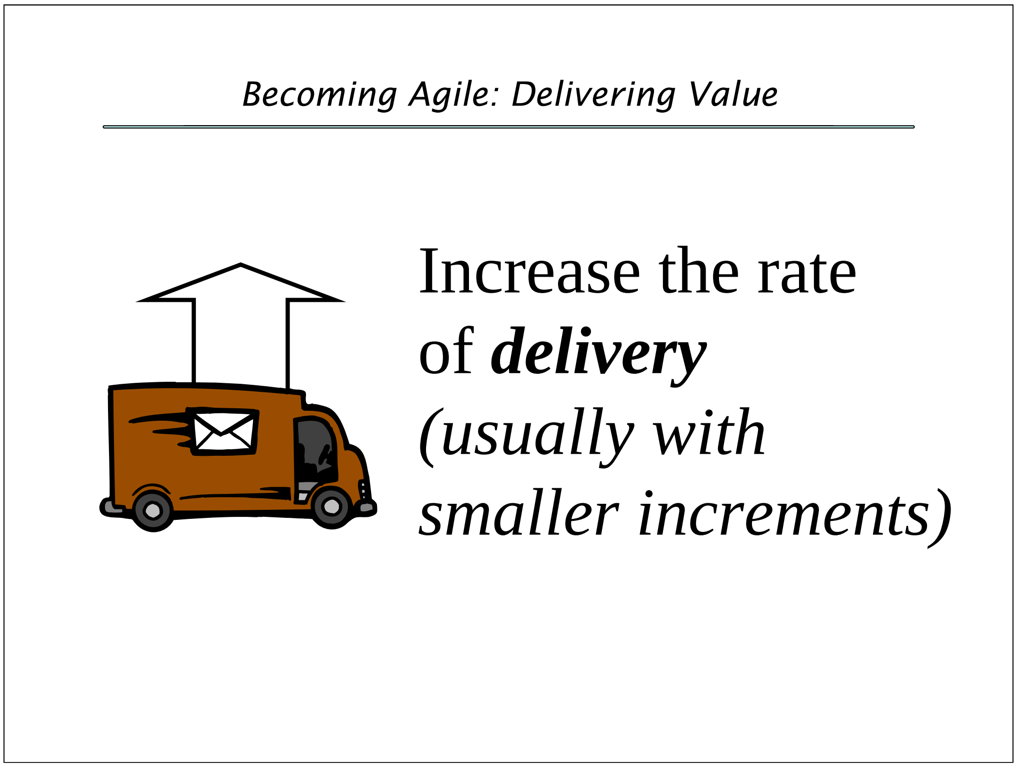
QA Becoming Agile: Shifting Roles

from last line of defense…

…to team support

QA Cannot Be Agile in Isolation

The practices described in this talk don't work when implemented in isolation. As we've seen, traditional test practices evolved to support traditional phased development processes. If you're in a traditional environment, stick with the traditional test practices: they're a better fit for your context.



Becoming Agile: Delivering Value

Increase the rate of ***delivery*** *(usually with smaller increments)*

Increasing the Rate of Delivery

Life in an un-agile company:

“How long until we can deploy the new Mobile client?” asked Jamie, the Product Manager. “I heard from Jill that the coding is done.”

“Yes, the Mobile features are coded,” responded Gerry, the Development Manager. “But we can’t deploy for another six months.”

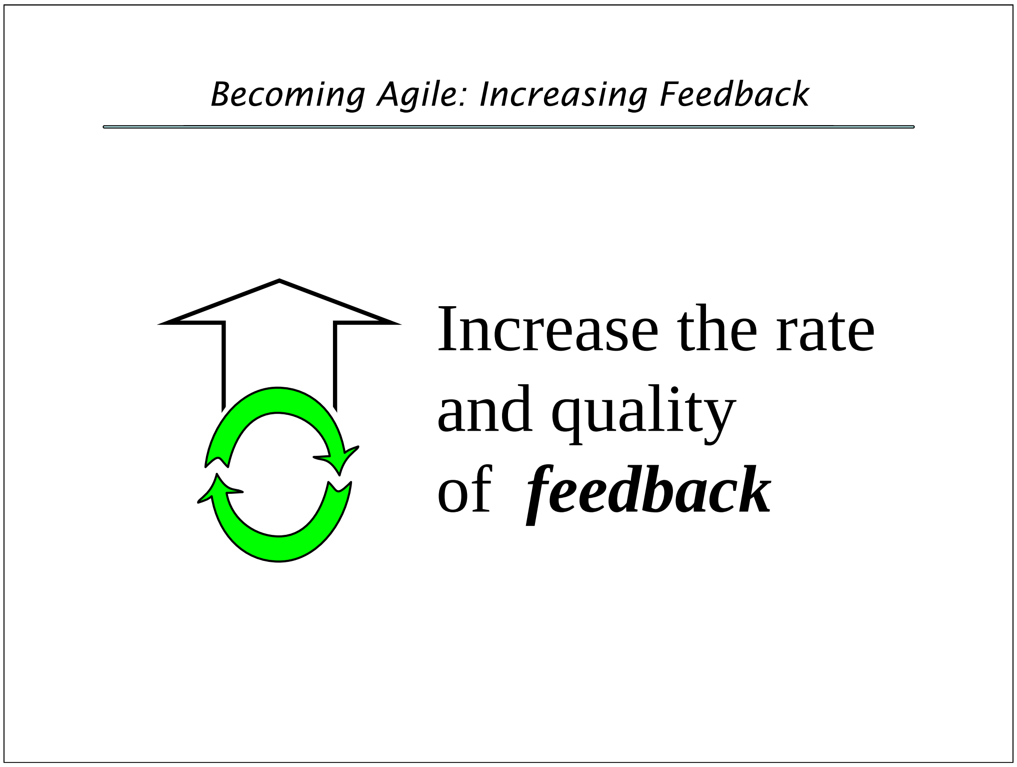
“Six months?!?” Jamie’s jaw dropped. “Why?!”

“Because the rest of the release won’t be ready for another six months.” Gerry replied. “That’s the soonest anything can go out.”

Six months is an eternity in today’s marketplace. In six months, the competition may have leapfrogged the current solution and snagged the majority of the market share. Speed is of the essence.

One way some organizations expedite the release of critical fixes or features is with a patch release. But patches come with a high cost. Usually people have to be pulled off their regular work to do a patch, test systems must be reset, and work on other releases comes to a screeching halt. Once the patches are done for a given release, the changes must be merged into one or more additional code lines. Code base management becomes a nightmare.

Agile teams use smaller increments and just-in-time collaborative planning to ensure critical changes can be released sooner. If something is important, it can be moved into an earlier release or iteration. It may displace another feature that isn’t as important, but it won’t disrupt an entire release. The result: faster delivery.



Becoming Agile: Increasing Feedback

Increase the rate and quality of ***feedback***

Shortening Feedback Loops

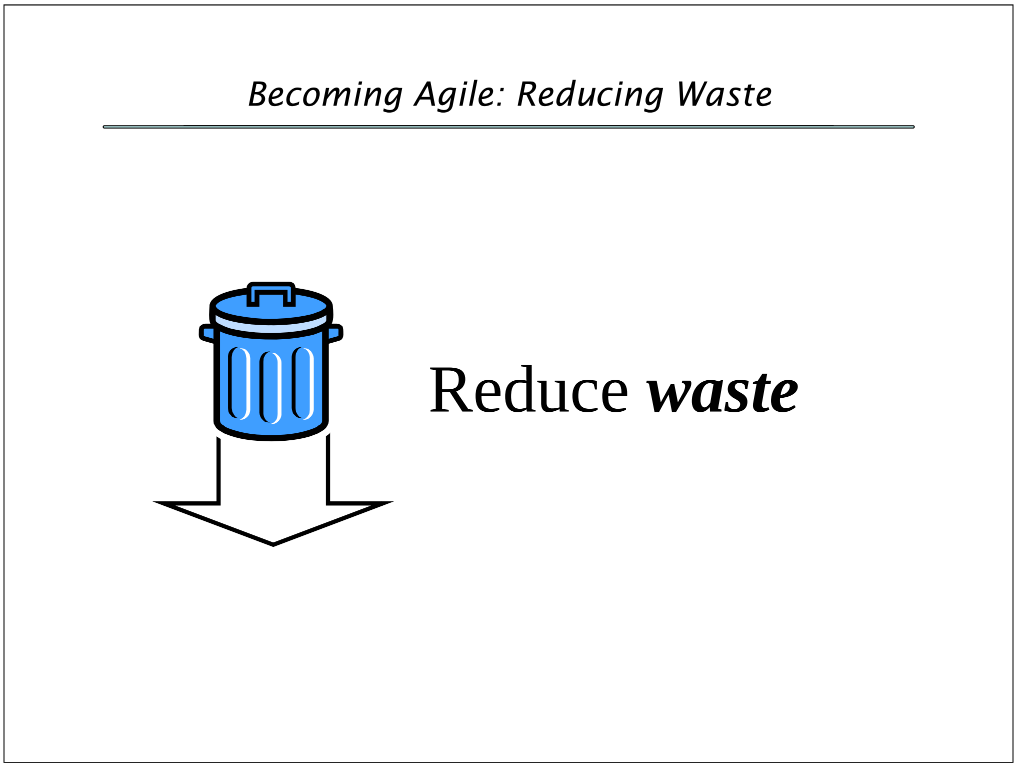
How long does the team have to wait for information about how the software is behaving? Measure the time between when a programmer writes a line of code and when someone or something executes that code and provides information about how it behaves. That's a feedback loop.

If the software isn't ready to test 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 check the behavior of the system end-to-end. They're typically run on checked in code on an ongoing basis, providing feedback in an hour or so.
- 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.

It's no wonder Agile projects favor automated tests. Though manual testing, particularly manual exploratory testing, is still important, even on Agile projects.



Becoming Agile: Reducing Waste

Reduce ***waste***

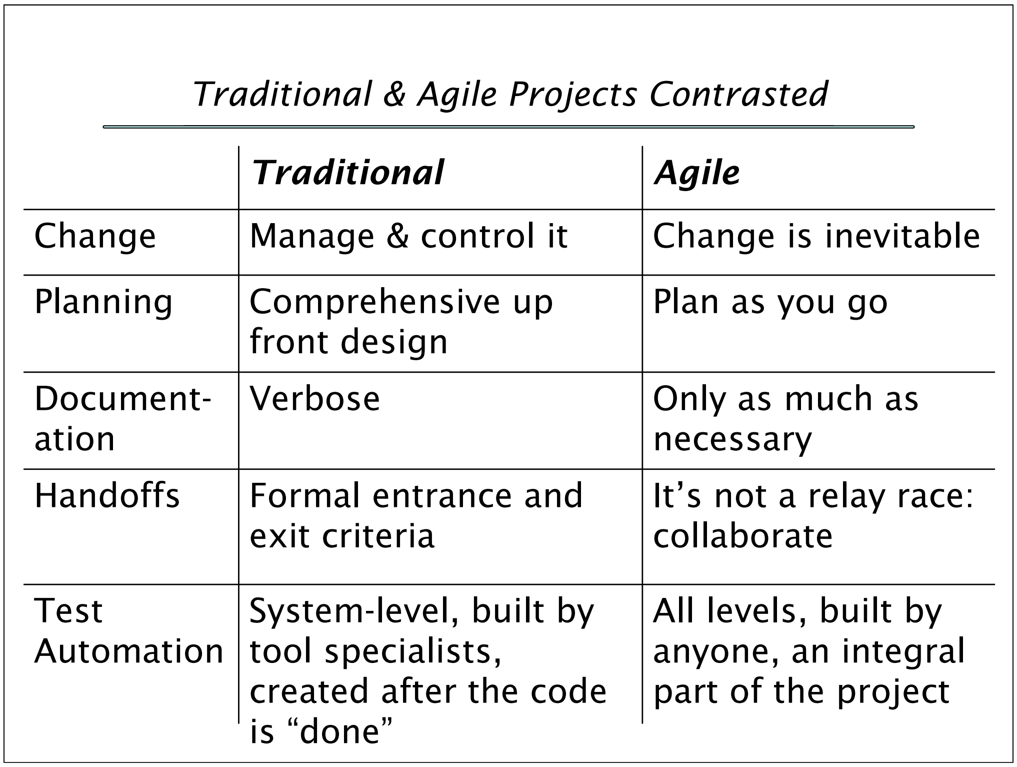
Waste in Software Development

In Lean terms, waste is anything that does not add value for the customer. Waste is not always something physical that is thrown away; inefficiencies are a source of waste as well. For example, “motion” is considered waste in manufacturing terms.

Software waste can include:

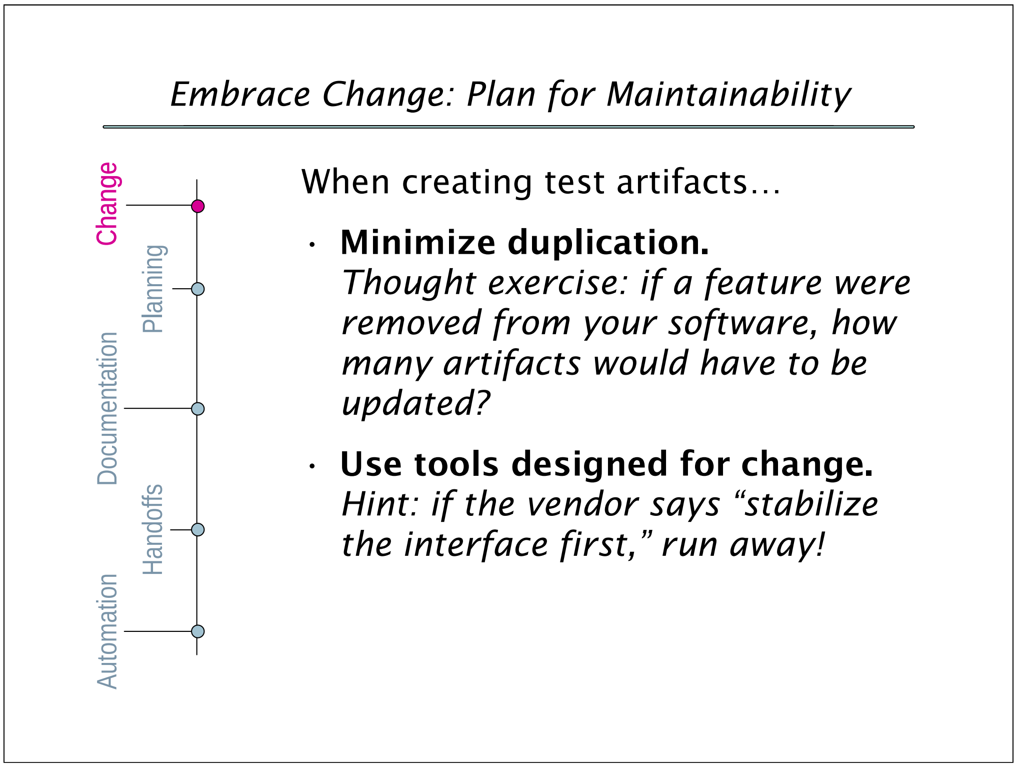
- Half-done features that are ripped out prior to release
- Unneeded features
- Excessive or unmaintained internal documentation
- Rework such as bug fixes/patches
- Invalidated test results such as test results for an invalid configuration or data set
- Time spent waiting for information, deliverables, decisions, or anything else that blocks progress

See *Lean Software Development* by Mary Poppendieck and Tom Poppendieck for more information on reducing waste on software projects.



Traditional & Agile Projects Contrasted

	Traditional	***Agile***
Change	Manage & control it	Change is inevitable
Planning	Comprehensive up front design	Plan as you go
Documentation	Verbose	Only as much as necessary
Handoffs	Formal entrance and exit criteria	It’s not a relay race: collaborate
Test Automation	System-level, built by tool specialists, created after the code is “done”	All levels, built by anyone, an integral part of the project


Change
Planning
Documentation
Handoffs
Automation

Embrace Change: Plan for Maintainability

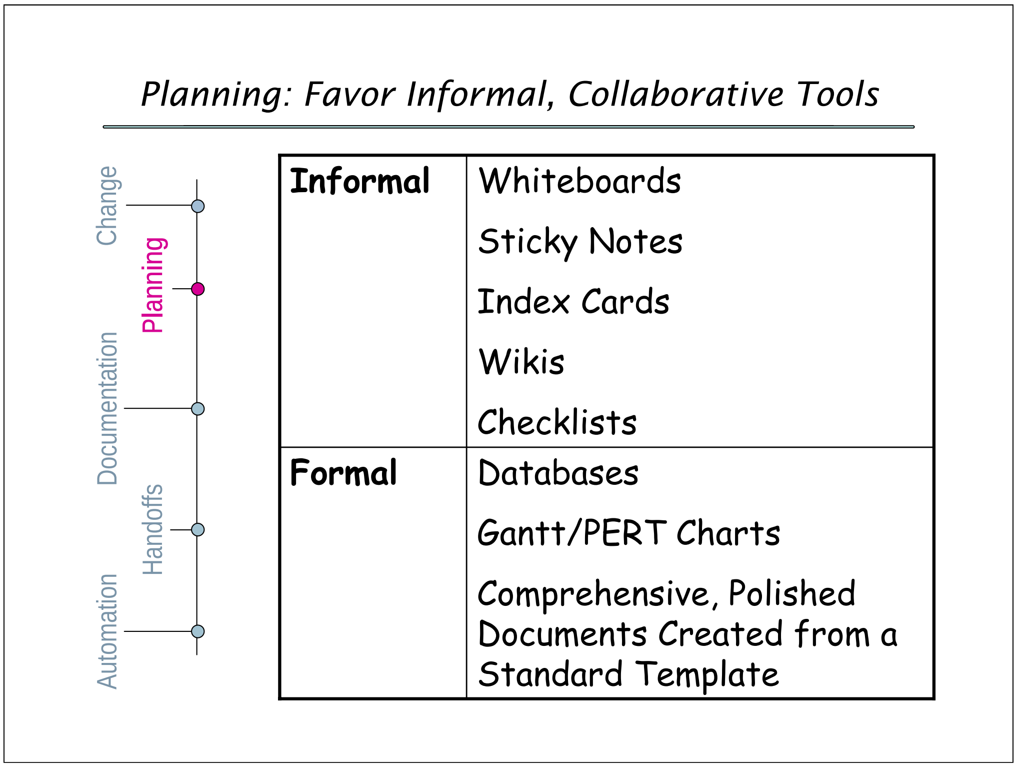
When creating test artifacts…

- **Minimize duplication.**
  Thought exercise: if a feature were removed from your software, how many artifacts would have to be updated?
- **Use tools designed for change.**
  Hint: if the vendor says "stabilize the interface first," run away!

A Traditional Tester on an XP Project

A tester recently commented to me, "I'm working with an XP team and they're driving me nuts! How am I supposed to design, document, and execute a full set of tests for the product in every 2 week iteration?!? They don't give me the specification until I nag them about it! And then in the next iteration, everything changes!"

This is a tester struggling with embracing change. He wants everything laid out neatly before he begins his analysis. He wants to do his test design in one fell swoop. He's having trouble adjusting his practices for his new context.



Planning: Favor Informal, Collaborative Tools

Change
Planning
Documentation
Handoffs
Automation

Informal	Whiteboards Sticky Notes Index Cards Wikis Checklists
Formal	Databases Gantt/PERT Charts Comprehensive, Polished Documents Created from a Standard Template

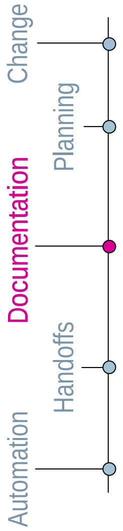
A little planning is good. More is not better.

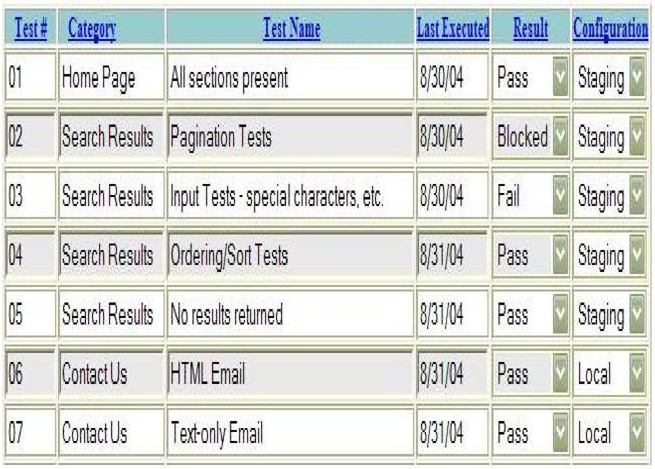
Plan for the current iteration. Design tests for the features or stories to be done in the near term. Speculative planning means rework.

Use catalogs of reusable tests. Don't redesign the same tests over and over. Instead, keep reusable checklists.

Keep an up-to-date, prioritized risks list. What kind of information are the testers looking for? The risks list covers it.

A Lightweight Example: Wikis





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

Built in Twiki with the Table plugin installed. See http://www.twiki.org

Test Documentation and Test Tracking All in One

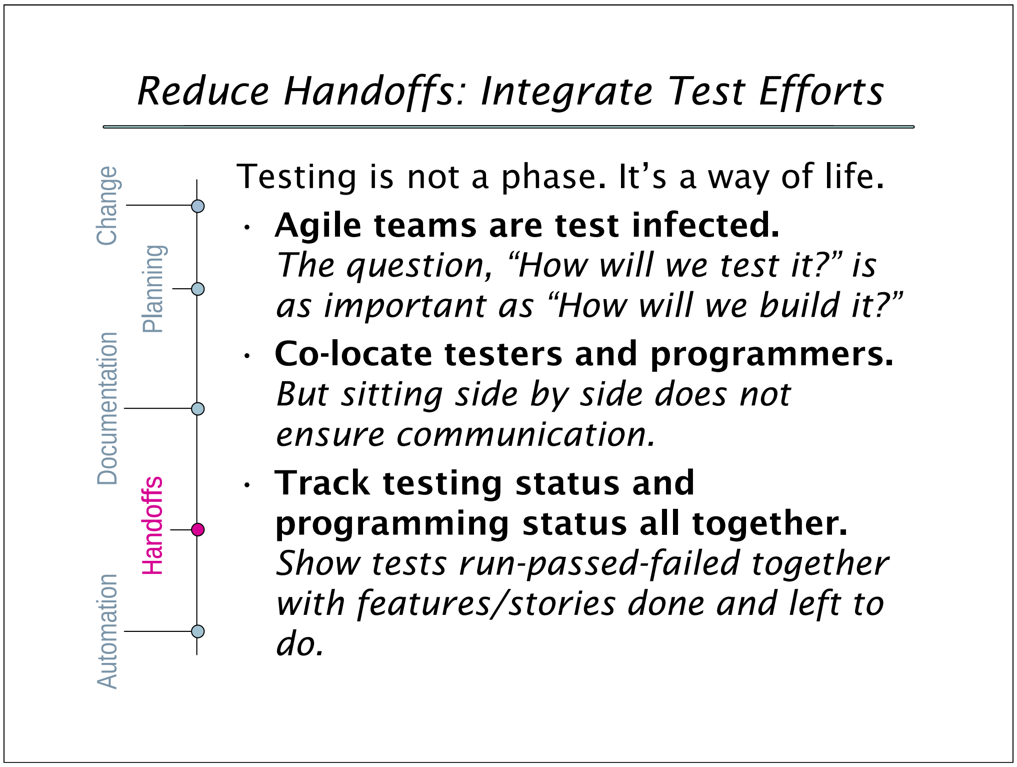
This Wiki page provides an example of leveraging one document for multiple uses. Here we see both the test design and the test results tracking all in one place. Reporting test status on this project took about 15 minutes total per week.

Tips for Lightweight Documentation

Capture the essence, not the details. Step-by-step instructions cost time without providing value (usually).

Point to other project documents. If it’s in the user guide, requirements, specs, etc., leave it there.

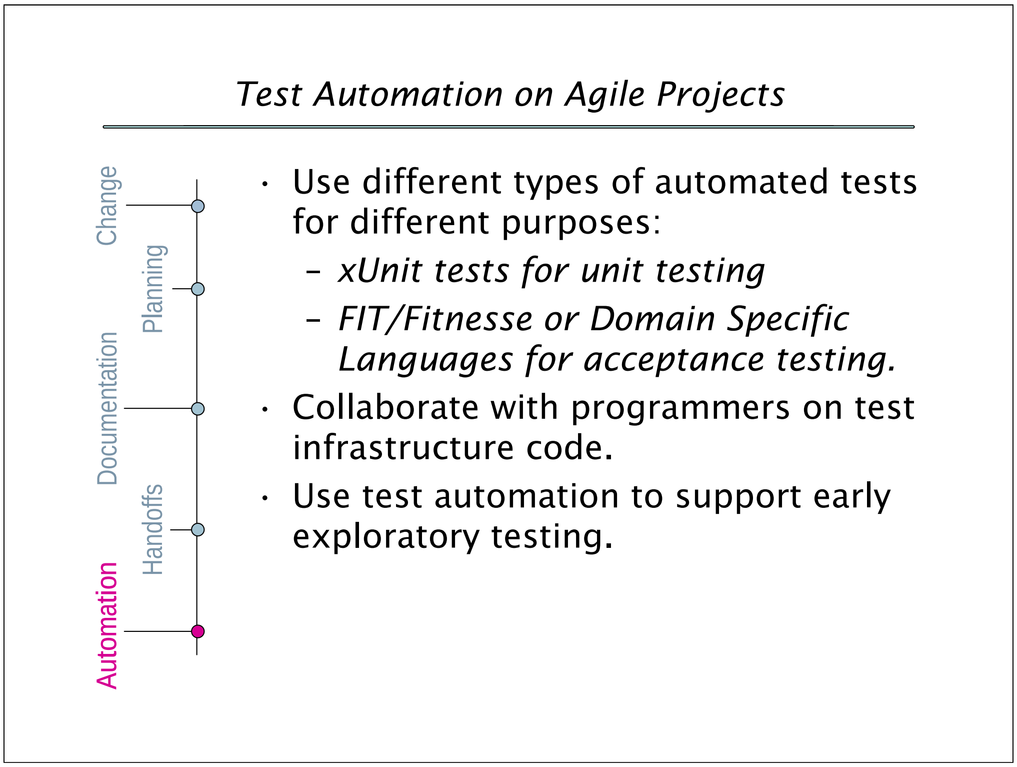
Centralize generic tests in a checklist. Try this: count the number of times common condition, like invalid dates or null strings, are documented in the test docs. More than once is too many.


Change
Planning
Documentation
Handoffs
Automation

Reduce Handoffs: Integrate Test Efforts

Testing is not a phase. It’s a way of life.

- **Agile teams are test infected.** *The question, “How will we test it?” is as important as “How will we build it?”*
- **Co-locate testers and programmers.** *But sitting side by side does not ensure communication.*
- **Track testing status and programming status all together.** *Show tests run-passed-failed together with features/stories done and left to do.*



Test Automation on Agile Projects

Change
Planning
Documentation
Handoffs
Automation

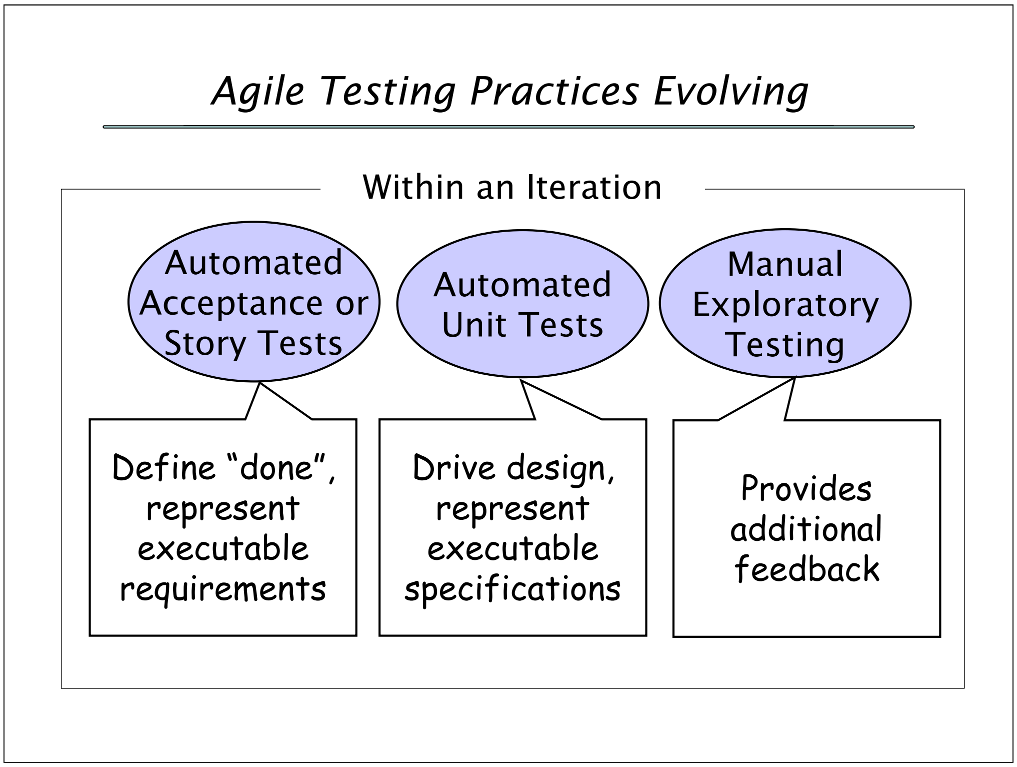
- Use different types of automated tests for different purposes:
  - *xUnit tests for unit testing*
  - *FIT/Fitnesse or Domain Specific Languages for acceptance testing.*
- Collaborate with programmers on test infrastructure code.
- Use test automation to support early exploratory testing.

The Difference Between Unit Tests and Acceptance Tests

Teams that attempt to substitute unit tests for acceptance tests, or vice versa, often run into trouble. These are two different types of tests, seeking answers to two different types of questions. Unit tests answer the question, “does the code do what the programmer intended?” while acceptance tests answer the question, “does the system do what the Customer wants?” Unit tests detect unintended change at a granular level (function, object, method). Automated system or acceptance tests should cover end-to-end sequences. Don’t substitute one for the other.

Leveraging Test Code for Exploratory Testing

Traditional test wisdom says we can’t start testing a feature until it’s accessible from an external interface (like a GUI). But we don’t have to wait. Test automation can facilitate manual exploration.



Agile Testing Practices Evolving

Within an Iteration

Automated Acceptance or Story Tests

Define "done", represent executable requirements

Automated Unit Tests

Drive design, represent executable specifications

Manual Exploratory Testing

Provides additional feedback

Unit Testing

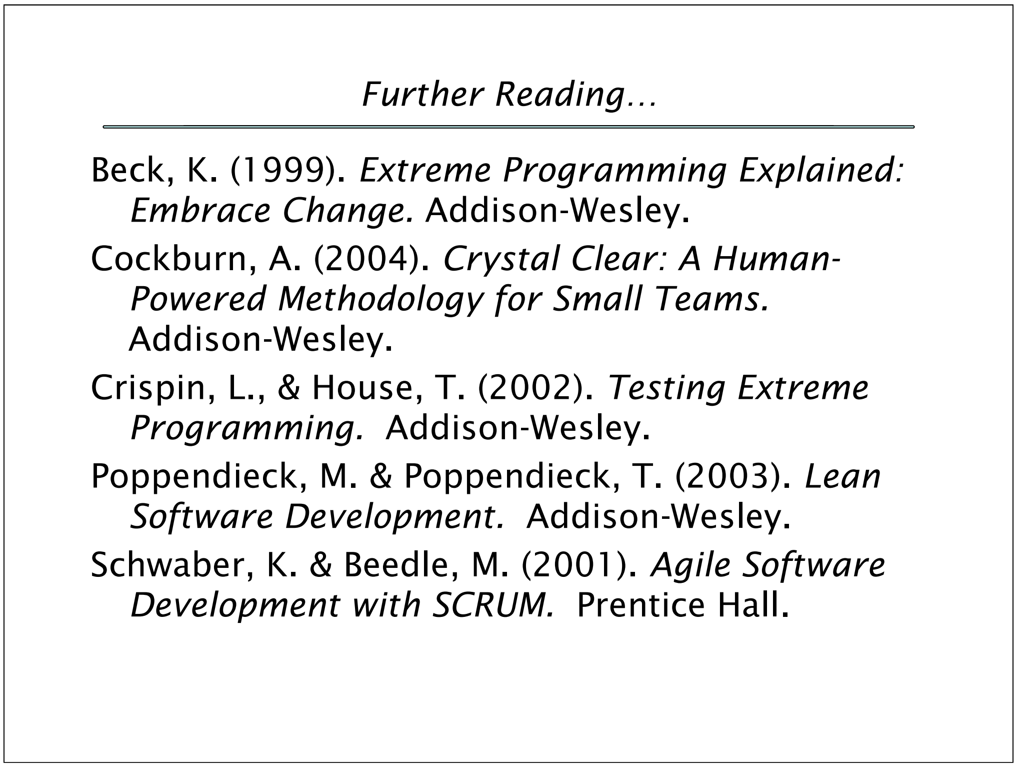
- Done by developers, usually with an xUnit framework, often as a result of practicing Test Driven Development (TDD)
- Supports the development process
- Unit test suites represent executable specifications

Automated Acceptance Testing

- The result of a collaboration between the developers and business stakeholders
- Often implemented in a FIT-like framework or Domain Specific Language
- Acceptance test suites represent executable requirements

Exploratory Testing

- Provides additional feedback and covers gaps in automation
- Necessary to augment the automated tests



Further Reading…

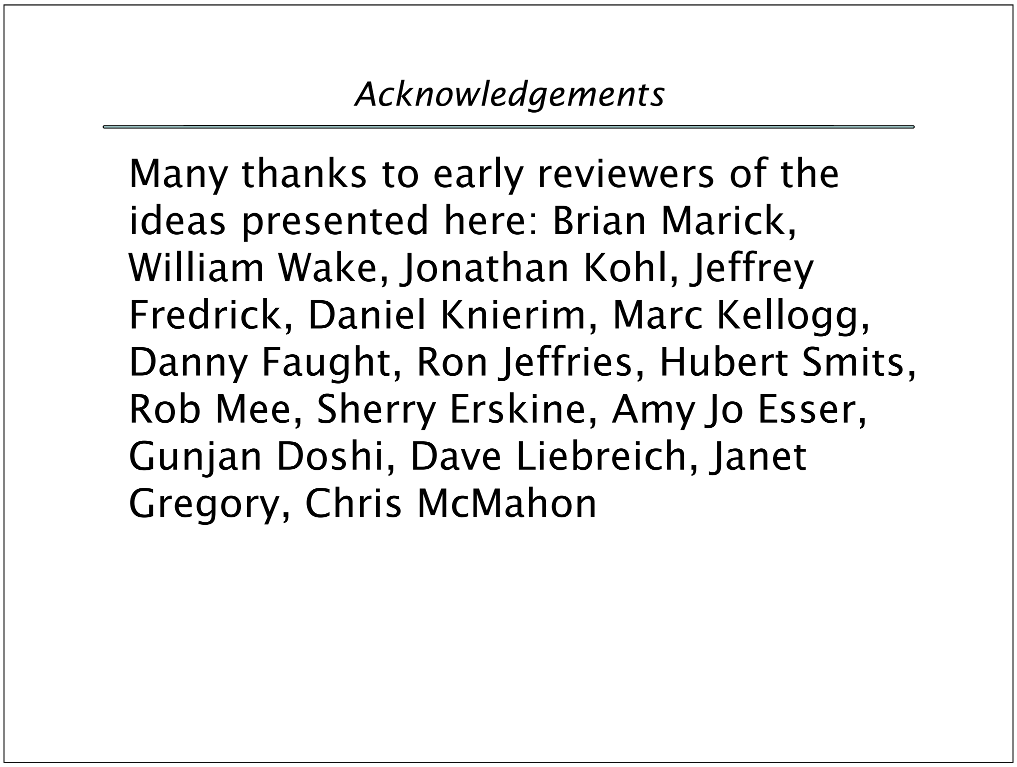
Beck, K. (1999). *Extreme Programming Explained: Embrace Change.* Addison-Wesley.

Cockburn, A. (2004). *Crystal Clear: A Human-Powered Methodology for Small Teams.* Addison-Wesley.

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

Poppendieck, M. & Poppendieck, T. (2003). *Lean Software Development.* Addison-Wesley.

Schwaber, K. & Beedle, M. (2001). *Agile Software Development with SCRUM.* Prentice Hall.



Acknowledgements

Many thanks to early reviewers of the ideas presented here: 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