Testing

Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often.

Testing does not improve quality; it exposes it.

Quality is never an accident; it is always the result oh high intention, sincere effort, intelligent direction, and skillful execution. - William A. Foster

==

Dr. W. Edwards Deming, an American statistician who led the quality movement in Japan (and later in America), spent much of his time trying to convince people that most quality problems are "in the process, not the person." For most of his 60+ year career, he promoted his 85/15 rule, based on his experience that 85% of problems were *built into* the way work was done (and hence under the control of management). Only 15% of the problems, he said, were really the fault of individual employees.

Most frontline employees had no trouble accepting Dr. Deming's assertions. After all, they were the people who paid the price for a lack of training, poor equipment, little communication, and unrealistic goals. In short, they worked under conditions that *guaranteed* poor quality. It was often *managers* who resisted Dr. Deming, because they were trained to find "who to blame" when something went wrong.

In the last few years of his life, Dr. Deming admitted his 85/15 ratio was probably wrong. More than likely, he said, it's 96% of problems that are built into the work *system*. Individual employees, he concluded, could only control perhaps 4%!
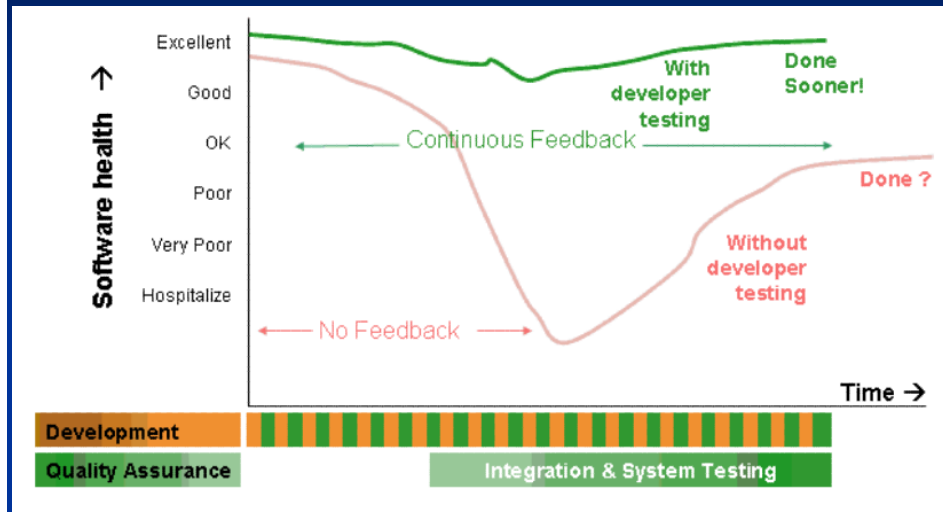
Extra problem:

In some IT organizations,

       Programmer performance was measured by lines of code delivered,

       Testing performance was measured by defects found

         So neither group was motivated to reduce the number of defects prior to testing.
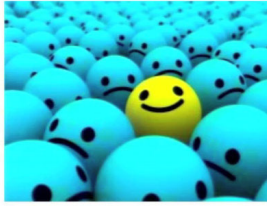
Traditional vs. Agile Testing

How long do companies free code before releasing production grade? A non-agile company will typically free allocate last 50% of sprint for code freeze. A typical company (some agile practices) will allocate 30% of sprint for code freeze. The best companies (agile implemented correctly) will generally allocate less than 10% of their sprint time for code freeze.

Why use Testers?

The path to Happiness contains many forks.

'Happy Path' test: one that demonstrates a user goal succeeding

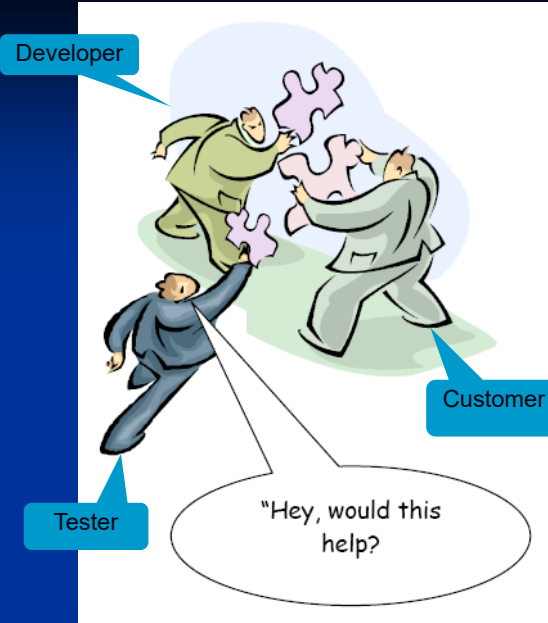A developer knows that a 'Happy Path' test is necessary

A tester knows that a 'Happy Path' test is not sufficient

Traditional view of Testing

Tips for those new to the test agile role

- Make sure that you attend the sprint planning meetings and wear your 'Business Analyst' hat – make sure you gather requirements that will enable you to write short test scripts and therefore test each backlog item and task. If need be arrange (with the Scrum Master and Product Owner) a follow-up meeting to clarify the requirements.

- Don't be shy in raising impediments – it's the Scrum Masters job to ensure that all impediments are removed so that "the team" can successful carry out their work. Impediments can include "I need more details on how this function should work."

- Review, on a regular basis, the product backlog and create and maintain a query log that maps each backlog item – this will help you when your in the sprint planning meeting to ask the right questions at the right time.

- Keep a defect log during each sprint - add those defects that are not cleared at the end of each sprint to the product backlog.

Defect Detection vs. Prevention
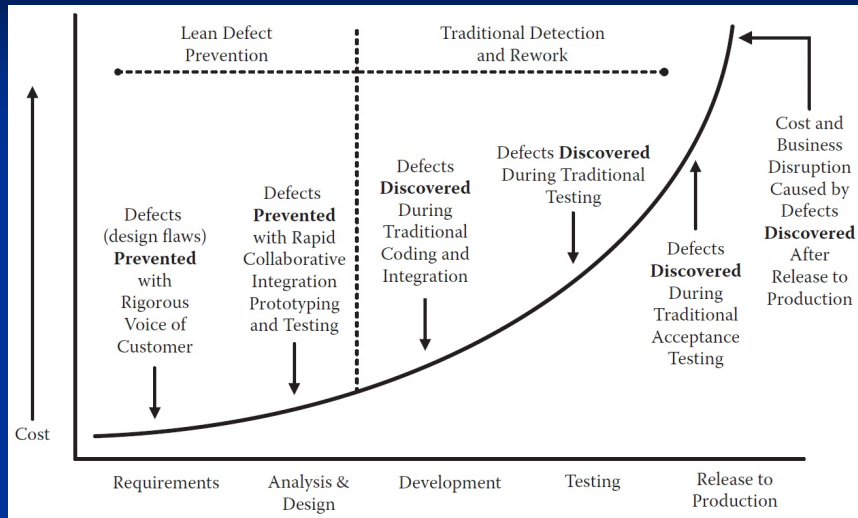
How would you prefer to find out about a serious defect?
• From your customer OR
• From your QA department OR
• Find it yourself.

Agile teams choose the third option.

In a defect detection culture, developers have abdicated the responsibility of testing and product quality to other groups, usually QA. This is wrong, because developers control the processes and practices used to produce the product and because control is essential to agility.

The primary difference between a defect detection and defect prevention culture is the attitude of developers and management toward testing (once again,

mindset). Developers must be expected to do everything in their power to prevent defects and to catch them before the software reaches QA and customers. It should be a point of pride that customers do not encounter defects and that defects found by QA are fixed as rapidly as possible. This does not mean that developers spend the bulk of their time doing manual testing, but it does mean that developers need to look at the task of software development as more than just writing code. The practices in this chapter are intended to help build a defect prevention culture, which is an important part of developing a professional attitude to software development.
--
I've had some developers tell me, "It's cheaper to fix defects after they're found than it is to prevent them." These developers are deluding themselves and confusing cheaper with easier: It certainly seems easier to concentrate on getting features out the door, then worrying about fixing them up later. But what these people miss is that customers find many of the defects, and that is extremely expensive for your company and for your customers. Even more important, however, is that having to fix defects later causes the team to lose control of the project, and this loss of control directly impacts the ability to be agile.

Defect prevention in a chemical manufacturing plant is taking the pumps offline periodically to perform preventive maintenance. Pumps that are regularly maintained are less prone to breaking down, and by pro-actively maintaining their equipment, the plant employees are able to avoid running from one crisis to another. They are in control of the situation, not victims of circumstance. A defect prevention mindset in software development should also result in being more in control than purely responding to events.

Defect prevention has real return on investment. Let's assume that the same number of design and coding errors are produced in a defect detection and prevention culture. In a defect detection culture, the majority of defects are found through manual testing and after code has been integrated into the product. On the other hand, in a defect prevention culture, the emphasis is on finding defects before integration into the product. The predominance of low-value manual testing and elaborate defect tracking systems in defect detection have a very real cost, as does the fact that there is a time delay between when a defect is introduced and when it is fixed. By contrast, in a defect prevention environment, the result of thorough automated testing, high-value manual testing, and fewer defects that reach customers more than offsets the cost of spending extra time in team collaboration and writing automated tests. An elaborate financial model could be developed, but hopefully a quick mental exercise should do.

A Defect report is only an opinion, until a stakeholder decides what action to take from it.

Comparing V-Model
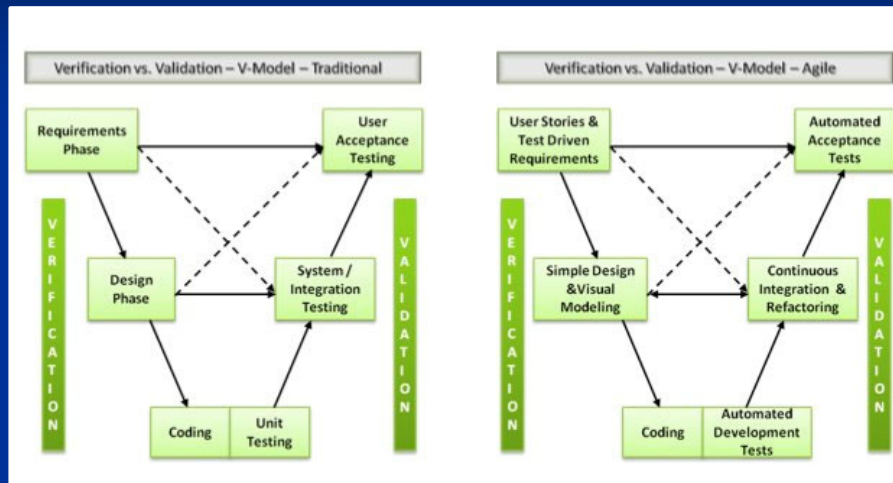
Compare with Traditional Testing

- Goals of Development Team and Testing Team
- When do we make sure that the System really runs?
- When are tests written and executed?
- How do we manage defects?

Agile policy: Zero post-iteration bugs. New features never accepted with defects

Traditional: Development team and Testing team have contradictory goals.

Agile: Both teams work together to ensure quality.

When do we make sure that the System really runs?

•Traditional: At the end of project

•Transitional: A few key milestones

•Agile: The system always runs

When are tests written and executed?

•Traditional: After development is complete, at the end of project.

•Transitional: After development, each iteration

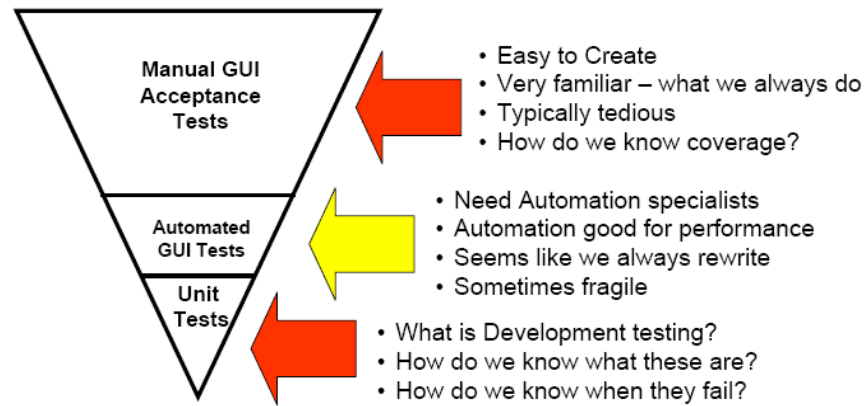•Agile: No code is written except in response to a failing test

How do we manage Defects?

Traditional, Transitional : Identified during test and logged in a defect tracking tool.

Agile: Zero post-iteration bugs; new features never expected with defects.

See: http://www.scrumalliance.org/articles/392-agile-testing-key-points-for-unlearning

Traditional Testing

Manual GUI Acceptance Tests
- Easy to Create
- Very familiar – what we always do
- Typically tedious
- How do we know coverage?

Automated GUI Tests
- Need Automation specialists
- Automation good for performance
- Seems like we always rewrite
- Sometimes fragile

Unit Tests
- What is Development testing?
- How do we know what these are?
- How do we know when they fail?

For an application of size 100KLoc, expect to write anything between 150KLoc to 300KLoc of testing code.

Acceptance Testing: Tell us whether the system does what the customer expects.

Unit test include testing for integration i.e. Testing stored procedures, triggers, foreign keys, XML, Configuration files, flat files, testing for null columns, testing for checks, testing for columns in the database, SQL statements, etc. Anything that the application depends upon should be tested.

Agile Testing Pyramid
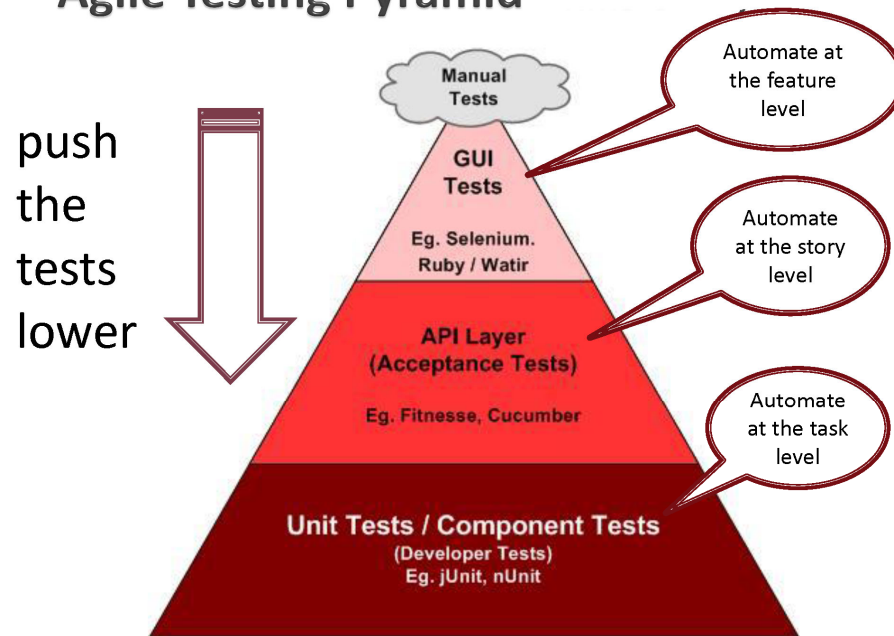
Common types of Testing

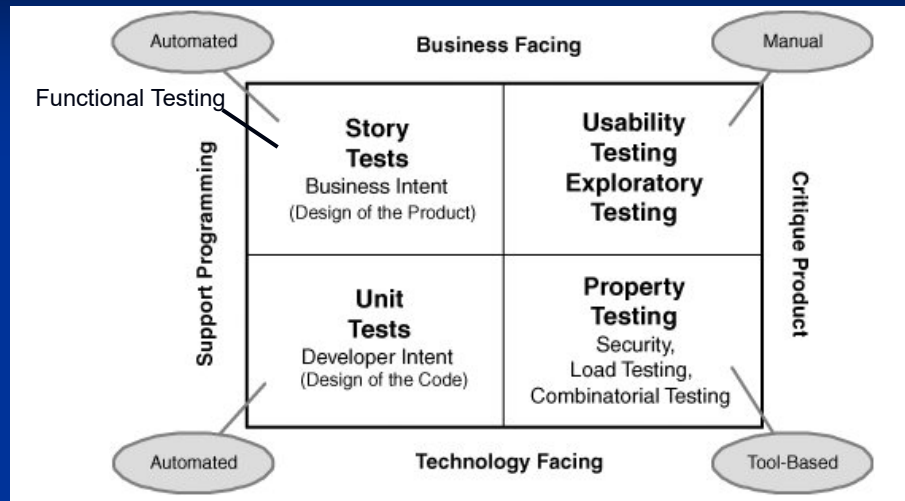7/9/2022 12:38 PM vijaynathani.github.io Page 13

Story tests are also called as Acceptance tests. They involve function testing and UI testing. Here tools like Fit, Fitnesse and Selenium are used. (Consider using jwebunit if there is no javascript. It is better than httpunit)

Usability and exploratory tests are, by definition, manual tests. When a system passes its automated tests, we know that it does what it is supposed to do, but we do not know how it will be perceived by users or what else it might do that we haven't thought to test for. During usability tests, actual users try out the system. During exploratory tests, skilled testing specialists find out what the system does at the boundaries or with unexpected inputs. When exploratory tests uncover an area of the code that is not robust, a test should be written to show developers what needs to be done, and the test should be added to the test harness.

Customer test-driven development (CTDD), also known as story test-driven development (SDD), consists of driving projects with tests and examples that illustrate the requirements and business rules. What do I mean when I say 'customer tests'? Basically, anything beyond unit and integration (or contract) testing, which are done by and for the programmers, testing small units of code and their interaction. Customer tests may include functional, system, end-to-end, performance, load, stress, security, and usability testing, among others. These tests show the customers whether the delivered code meets their expectations.

Exploratory testing is scientific thinking in real-time. Puts test design and test execution together.

The tests that we did before inform the tests that we will do now. Those tests will inform the tests we will perform later.

Exploratory testing excels when

1) Imagination and creativity open up possibilities

2) There is little in the way of documentation

Exploratory testing deficiency

1) Less likely to find new revelations by going over the same ground. Pesticide Paradox

2) Humans get bored easily.

Example of Concordion / Fitnesse

CardType	Expiration	Number	valid()
Visa	05/05	4123456789011	true
Visa	05/23	4123456789012349	false
MasterCard	12/04	5123456789012343	true
MasterCard	12/98	5123456789012345	false
MasterCard	12/05	42	false
American Express	4/05	341234567890127	true

Testing for valid credit cards

Fundamental shift in thinking.

Time required for fit testing is 20% to 30% of the total time required to develop the user story.

Acceptance Testing with Concordion/Fitnesse

- Also called Customer tests or Functional Tests
- Serves as executable requirements documents
- Can be run anytime by anyone.
- Non-technical
- Build the RIGHT code

Defines how the PO wants the system to behave.

Specified by PO.

Enables the PO to verify that the system works as required.

Can be understood by non-technical people.
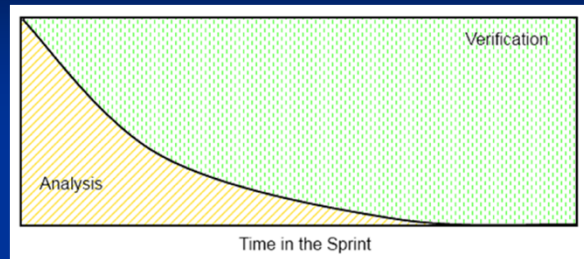
End-to-end testing is overrated.

OK for showing some path through the system is correct, poor at showing where it is incorrect.
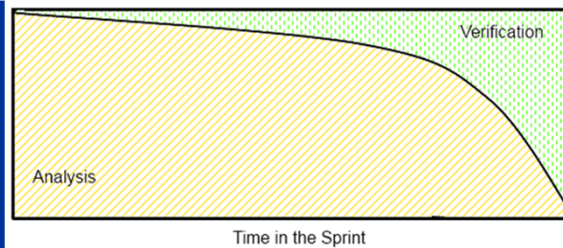
"Trying to test the depths of the code through the UI is like peering through the shower head to examine pipes in the basement."

Elisabeth Hendrickson

Time of Testers

Traditional

Verification

Analysis

Time in the Sprint

Agile

Verification

Analysis

Time in the Sprint

Although there are no titles like "Software Tester" or even "Software Developer" in Scrum teams, there are Development Team members with a range of skills that all work together to deliver an increment of functional software. Many of those Development Team members were originally hired as Software Testers and they bring a set of skills to the team that are some of the most necessary when delivering a done increment. That said, many software test professionals feel their job lies primarily in verifying functionality, rather than guiding it. This is a symptom of the waterfall way we've worked for so long. Development phase came first, and testing came after, thereby practically defining the role of software testers as a verification step.

Scrum calls for asynchronous execution, as do most agile methods. Yet, many Development Teams struggle how testing professionals can engage effectively during a Sprint. Common questions on the topic sound like these:

How can I test something that doesn't exist yet?

We automated all the testing, what do the testers need to do?

It's Acceptance-Test Driven Development (ATDD). Simply put, ATDD is way to develop code in which we write failing automated tests that read very carefully as requirements. You've heard this refrain before, and ATDD is a form of Test-First Development as well as a form of executable specification.

The key element of ATDD for Scrum Development Teams is that we can express requirements as failing tests, and do so at interesting levels of our application. The levels most teams find value in are at the Domain Conversation layer and at the UI layer.

Guess who is amazingly good at creating requirements like these. If you said testers, you win.

In my experience, identifying and capturing these scenarios is exactly where testers excel. In fact, they are typically better at it than developers, who (like me) tend to focus on the happy path. A Scrum Development Team can take great advantage of this skill by having some people on the team focus on creating acceptance criteria like those above as sets of failing automated tests. While a team may never get away from some component of manual verification, the amount of time spent doing this goes way down when we focus on building a regression harness from the requirements themselves.

Responsibilities and Tools

- Developer is responsible for white-box testing and performance profiling.
  - Rest of the testing is joint responsibility of QA and Developer
- Tools used
  - White-box unit testing – JUnit, MsTest
  - Black-box functional testing – Cucumber, SpecFlow
  - GUI testing – Selenium.
  - Performance Testing – Jmeter.

20% of the code is usually responsible for 80% of performance problems.

If CPU utilization above 80% for a long time, then usually it will cause other problems.

For Performance Testing,

Run it multiple times before coming to any conclusion.

Use database of about the same size as real world.

Try to replicate the real world environment to the extent possible but eliminate the network. i.e. Keep everything on the same network switch.

Avoid paid tools like LoadRunner, WinRunner, Silk, etc.

You cannot customize them.

We need to learn proprietary languages to use them.

They are usually not worth the money they charge.

QA and Developers

Week One

Role	Monday	Tuesday	Wednesday	Thursday	Friday
Developer	Coding	Coding	Coding/ Defect Fixes	Coding/ Defect Fixes	Coding/ Defect Fixes
QA/Tester	Write tests	Write test	QA/Testing	QA/Testing	QA/Testing

Week Two

Role	Monday	Tuesday	Wednesday	Thursday	Friday
Developer	Coding/ Defect Fixes	Coding/ Defect Fixes	Defect Fixes/ Design/Story Development	Defect Fixes/ Design/Story Development	Defect Fixes
QA/Tester	QA/Testing	QA/Testing	QA/Testing	QA/Testing	Acceptance Testing

TDD with unit testing is for building the code right. Acceptance Testing and QAs are for building the right code..

See: http://agilecommons.org/posts/993ce028e5

In the past few weeks I've been asked about and have been considering exactly how to fit QA and testing into a two week iteration.

•A primary concern of the folks I've been talking with is that QA's and testers on an agile team have nothing to do at the start of an iteration.

•The second concern is that we can't keep writing new code up until the last minute of an iteration if QA is to adequately test the code, and as such, what do the developers do at the end of the iteration.

Of course, the underlying concern in both of these cases is keeping the QA's and the developers effectively utilized during an iteration. Software quality always seems to boil down to a utilization/cost equation doesn't it? Well, after giving it some thought, I think I've come with a basic schedule for QA's and developers over a two week iteration.

So, let's take a walk through the schedule. On the first two days of the iteration, the developers get busy coding the stories in their backlog while the QA's start writing test cases/acceptance tests. The QA's should also be running these tests against the existing code base. This validates that the test harness is working correctly and that the new tests do not mistakenly pass without requiring any new code. Obviously, the new tests should also fail for the expected reason. This tests the tests themselves, in the negative: it rules out the possibility that the new tests will always pass, and therefore be worthless. OK, so that's day one and two.

Over the next five days, the QA's begin testing any testable code that has been completed. At the same time, the developers continue coding and also start working on fixing any defects picked up in the testing. Pretty standard days. Code-test-fix.

OK, we're deep into the iteration now. Days eight and nine see QA continuing to test all of the remaining code written during the iteration. Yes, **ALL** of the remaining code. At this point, the developers effectively stop writing "new" code and focus their energy on fixing all defects identified in testing. If the dev team has no defect work or find themselves with down-time while waiting for testing results, they can and should be helping the product owner develop and refine the user stories that are likely to be played in the next iteration. Additionally, developers can start considering the design aspects of the immediate upcoming stories and coordinating design decisions with the project architect.

Last day! Don't start scrambling now, you've got a demo and review later today. So, on Friday, the developers focus on bashing any remaining defects and making the code bombproof. We're shooting for zero defects here folks. The QA's are spending most of their time doing some final acceptance testing (as is the product owner). That's it. It's noon and it's time for your review and demo. You've tested and fixed everything so you and your team can demo with total confidence. No surprises for your team, you've built serious quality into your software!

So, there's my idea for a basic schedule that completely integrates QA and testing into your iteration. Now, this is only one way I think QA can be integrated into an iteration. There are probably dozens of other ways to do this and I think ultimately, the way teams work QA into their iterations really needs to be assessed on a case-by-case basis. So, I'd like to pose the question to agile teams out there: How do you currently fit QA/testing into your iterations?

Observations

- Automated unit testing is mandatory.
- Testing activities (excluding the programmer's unit testing and automatic builds), take up 20% to 50% of total development effort.

If Unit Test code coverage is less than 70%, then probably developers are not writing enough unit tests.

Code coverage of 100% is theoretical. ROI is usually bad above 95% code coverage.

Code coverage with automated tests in an Agile project is generally between 80 to 95%. This is probably the most important decision impacting quality

Testing only to end user requirements is like inspecting a building based on the work done by the interior designer at the expense of foundations, girders and plumbing.

Treat the daily build as the heartbeat of the project. If there's no heartbeat, the project is dead.