SSW-555: Agile Methods for Software Development

Software Testing

Prof. Jim Rowland
Software Engineering
School of Systems and Enterprises

Today's topics

- Overview of testing
  - Definition
  - Testing stages
  - Testing in Plan Driven and Agile methods
- When to integrate?
- Test-First (or Test-Driven) Development
- Evolution from Test-Driven to Behavior-Driven
- FitNesse – tool for creating acceptance tests

Software testing

The goal of software testing is to determine if the implementation meets its specifications

- Does the system do what it's supposed to do?

Testing and debugging are related, but different tasks

- Testing identifies problems – debugging fixes problems

Software testing

Unit Testing

- Test new features

Integration Testing

- Combine and test code from multiple developers

Regression Test

- Verify that new changes haven't broken previously working code

System Test

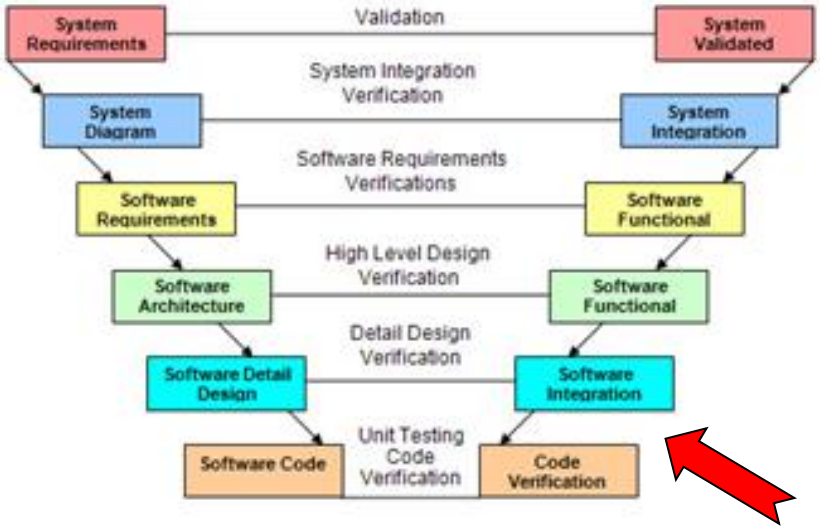
- Test the complete system
  - Functionality
  - Performance
  - Stress
  - Security

User Acceptance Test

- Verify that the system meets user's expectations

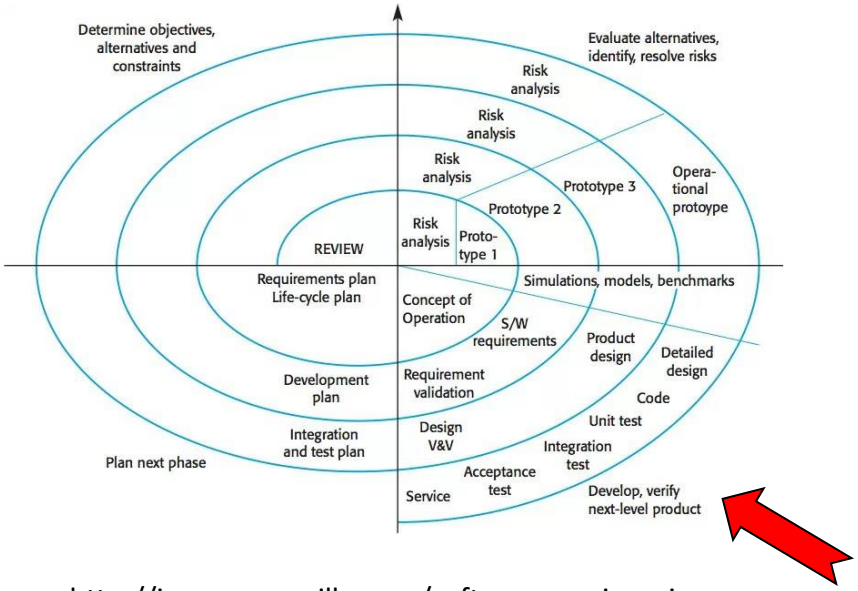
Traditional Software testing

Waterfall/V Model



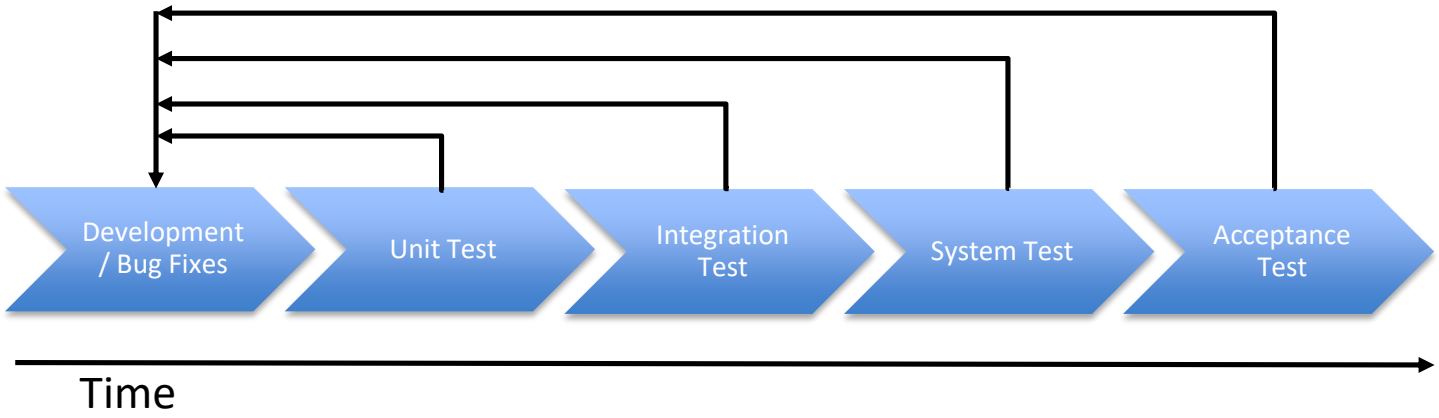
https://sites.google.com/site/advancedsofteng/software-acquisition/software-development-lifecycle-approaches

Spiral Model



http://iansommerville.com/software-engineering-book/web/spiral-model/

Traditional testing gates



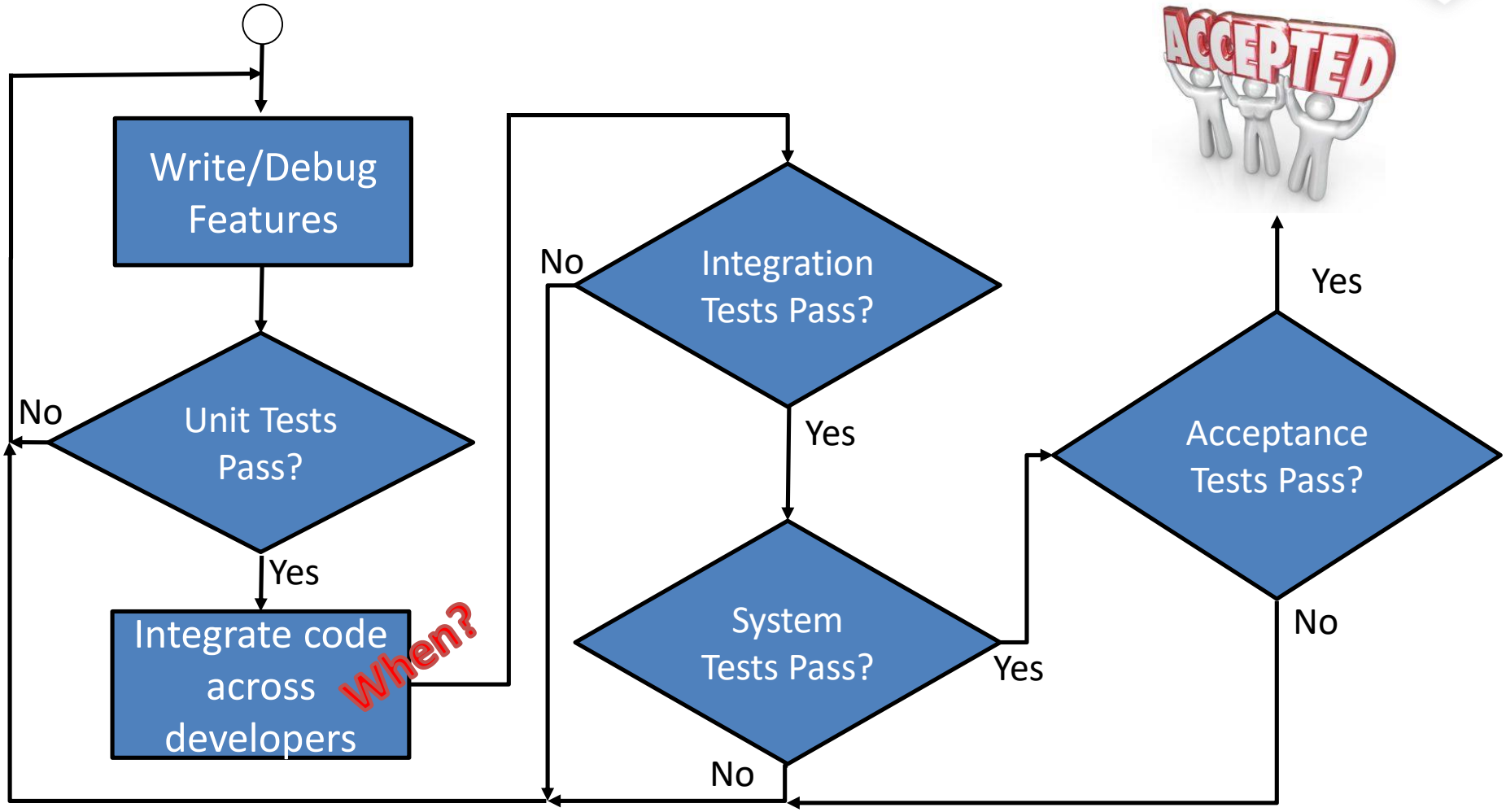
Testing is blocked until development is "complete"

Testing is performed by a separate test team

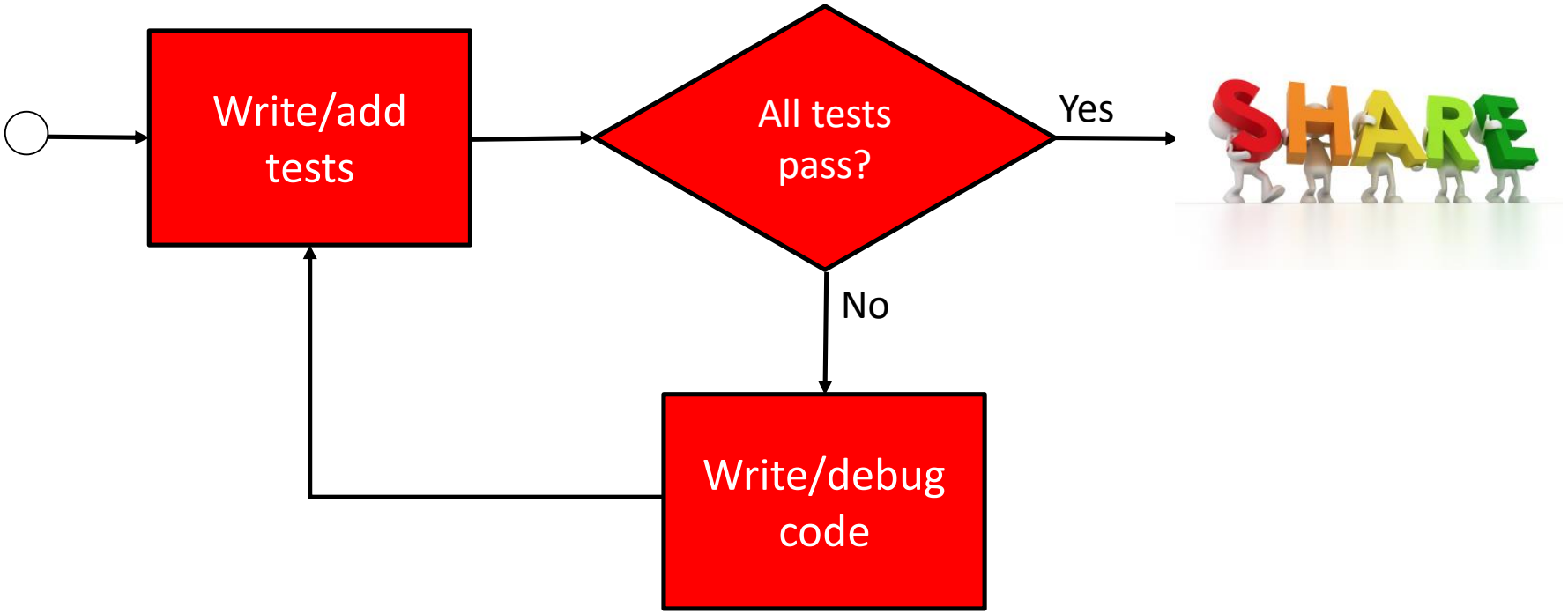
Developers might not be trusted to test their own code

Testing schedule may be compressed if development schedule slips

Traditional testing flow



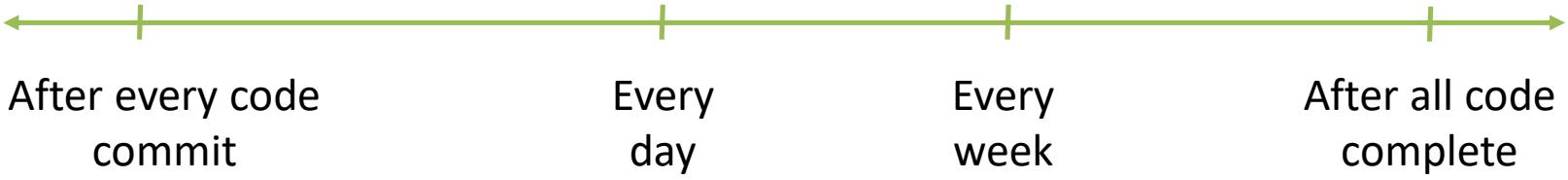
Agile testing flow: automated and continuous



When to integrate?

Integration combines code from multiple developers

How frequently? Continuously? Periodically?



Agile Methods

Traditional Methods

Functional vs Agile Teams

Functional teams hand off to next team, e.g. developers hand off to testers and back to developers…

Agile teams have fewer boundaries

"Whole team" approach to quality

Testing: Traditional and Agile

	Who?	What?	When?
Unit Testing: ensure that recently changed units work correctly	Developers	New features or bug fixes, code/branch coverage	After code complete, before integration
	Developers using automated test platforms	New features or bug fixes, code/branch coverage	Tests written before the code is written. Tests run while developing code
Integration Testing: ensure that components, potentially written by different developers, integrate cleanly and work together	Developers and/or testers	Test multiple modules from all developers	After unit test is complete on relevant modules
	Developers using automated test platforms with tests written during development	Test all modules from all developers	Continuously: at least once per day if not on every code check in using existing tests

Testing: Traditional and Agile

	Who?	What?	When?
Regression Testing: verify that changes haven't introduced new problems in already tested code	Test Team	All code, emphasizing new code	After integrating new features
	Developers using automated test platforms with tests written before code	Entire system or selected components	During integration with CI tools
System Testing/ Acceptance Testing: customer verifies that the system performs as expected and meets requirements	Test Team	Entire system	After code complete
	Developers, Testers, Customers with automated tests defined in User Stories	Entire system, including new features from last sprint	At the end of every sprint during Sprint Review and before every release

Software testing

Traditional Methods

Identify and react to quality issues

Agile Methods

Proactively reduce quality issues

Agile Testing Manifesto

Agile Testing

Testing is ***NOT*** a separate phase with Agile Methods

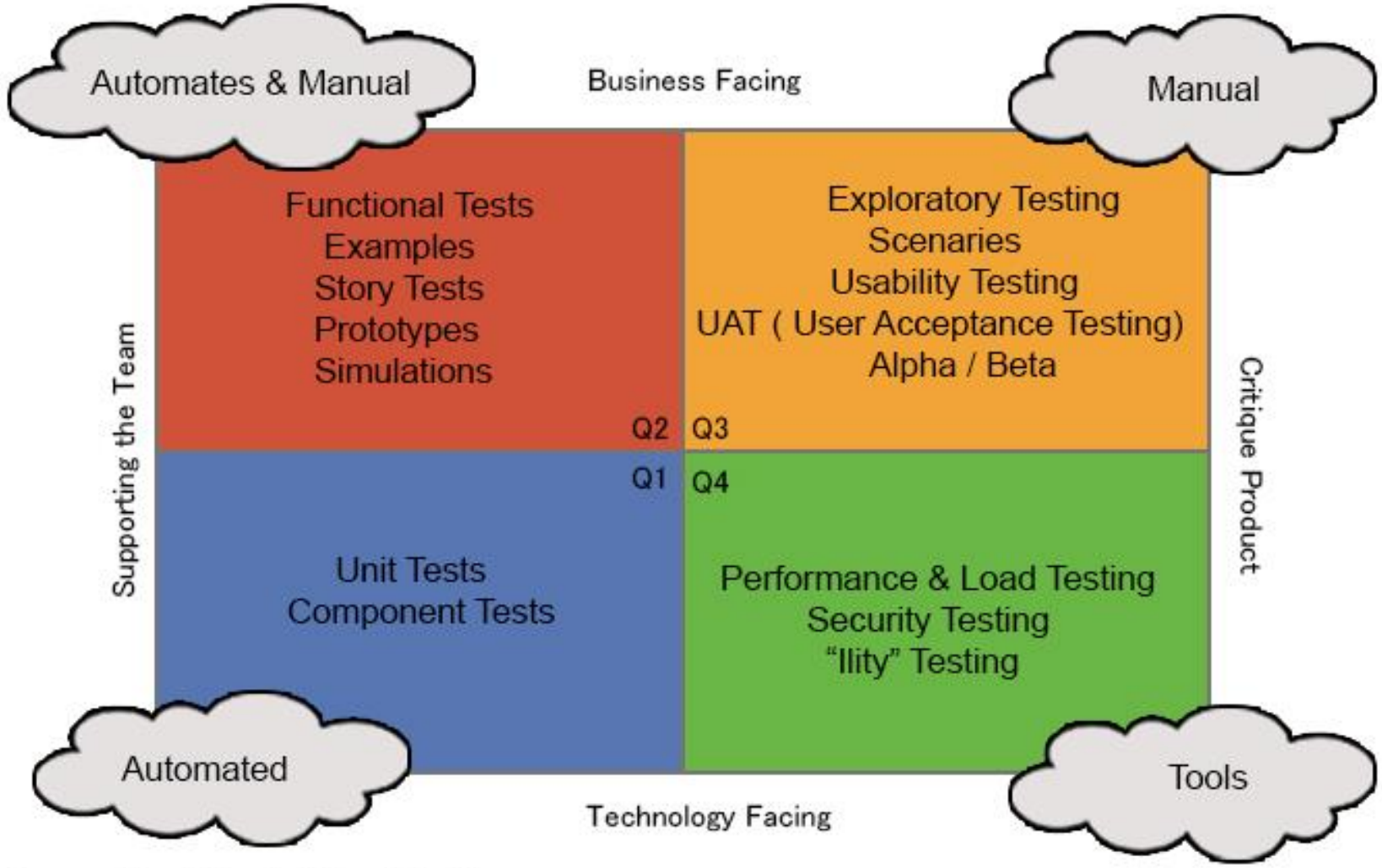
- Instead, testing is integrated into every step

Focus on ***automated testing*** and ***continuous integration***

- Find problems as quickly as possible
- Expect frequent changes, so facilitate them

Agile testing is not just a passing phase…

The Agile Testing Quadrants



Source: Lisa Crispin, Brian Marick

Technology facing tests/Supporting the team

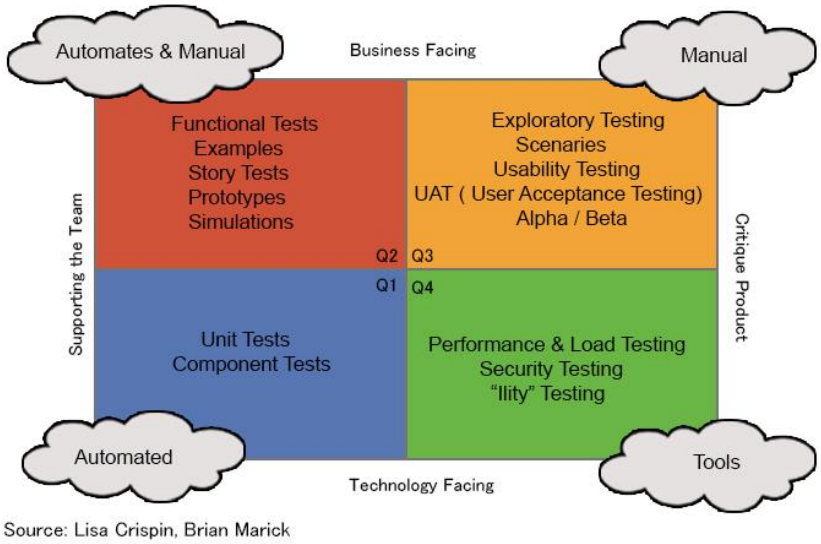
Q1: Verify functions and features perform as expected

Unit test

Highly automated, e.g. xUnit

Written by developers using TDD

Pair Programming, Configuration Management, Continuous Integration, etc. help to insure quality



Source: Lisa Crispin, Brian Marick

Business facing tests/Supporting the team

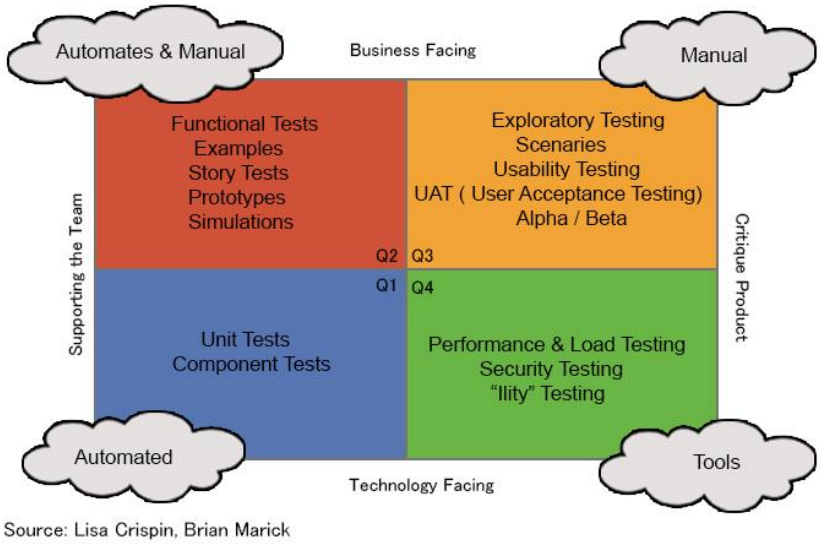
Q2: Verify business/customer-facing issues

Test desired features collected from customers

- Written when collecting user stories

Each user story includes "Definition of Done"

Verify that we're building what the customer wants and needs, e.g. Requirements and User Experience testing



Source: Lisa Crispin, Brian Marick

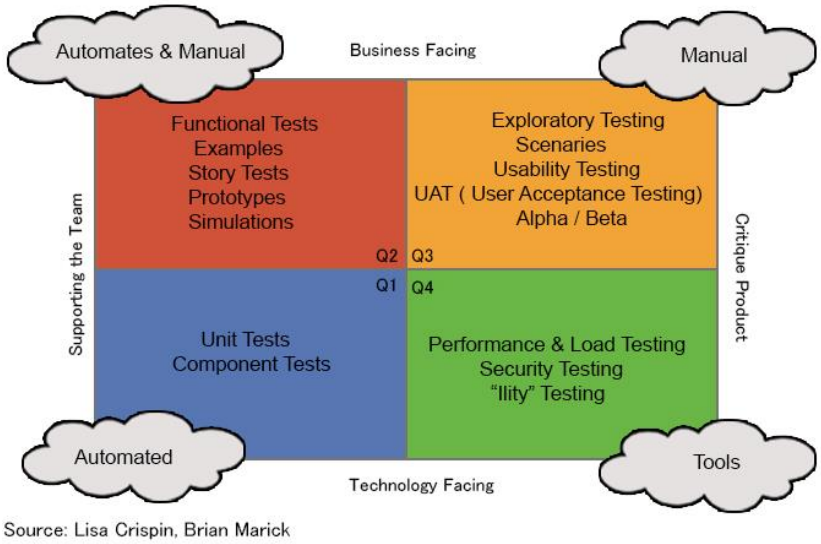
Business facing tests/Critique the Product

Q3: Verify business/customer-facing needs

Verify the working system against business needs, e.g. usability testing

Evaluate the system as an end user

Analogous to User Acceptance Testing but happens throughout the process, not just at the end

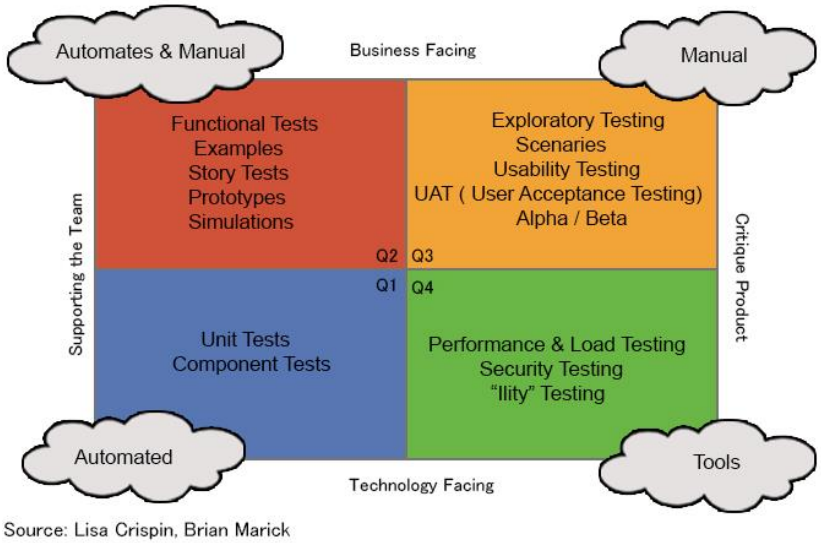


Source: Lisa Crispin, Brian Marick

Technology facing tests/Critique the Product

Q4: Verify technology needs

Verify the non-functional requirements, e.g. performance, load, security, reliability, data migration, scalability, …



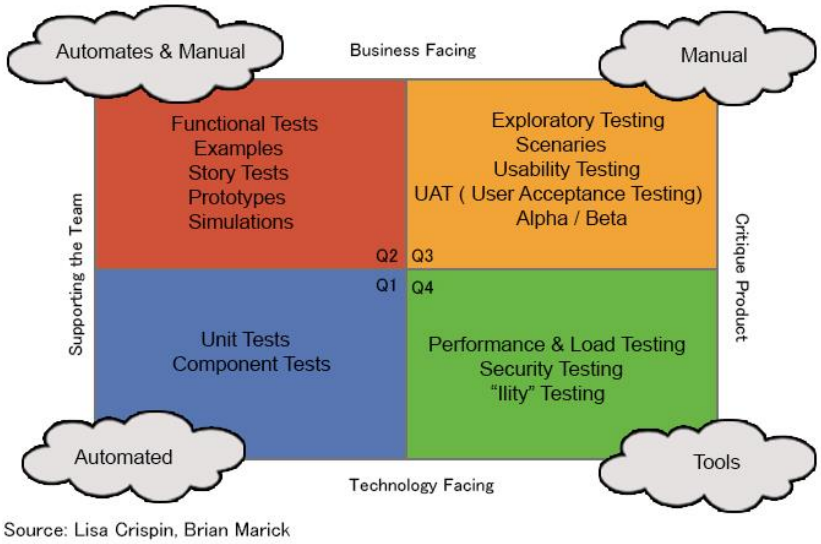
Source: Lisa Crispin, Brian Marick

Agile Testing Quadrants Summary

Agile Testing is ***not*** sequential with well defined gates

Use the Agile Testing Quadrants to guide testing effort throughout the development process

Consider each quadrant every sprint and adjust priorities



Source: Lisa Crispin, Brian Marick

What could possibly go wrong?

Agile testing has potential challenges:

- Changes happen quickly and problems may be overlooked
- Schedules for development and test are highly compressed
- Frequent changes to features makes testing challenging

Potential Solutions:

- Whole team approach to quality, including customer
- Automation (automated testing, Continuous Integration, …)

Customer bug reports?

Agile Methods' focus on automated testing and continuous integration helps to reduce, but doesn't eliminate bugs found by customers

Scrum adds new features each sprint

- How are bugs tracked?
- When are bugs fixed?

Add bugs as user stories to the product backlog

- Product owner prioritizes new features and bug fixes

Test-First or Test-Driven Development (TDD)

Motivation:

- Programmers don't write tests because:
  - They don't like to
  - They don't have time
  - They have "more important" things to do
- Result:
  - Code breaks
  - Debugging reduces productivity and doesn't improve testing
  - Still no tests

Alternative TDD Scenario

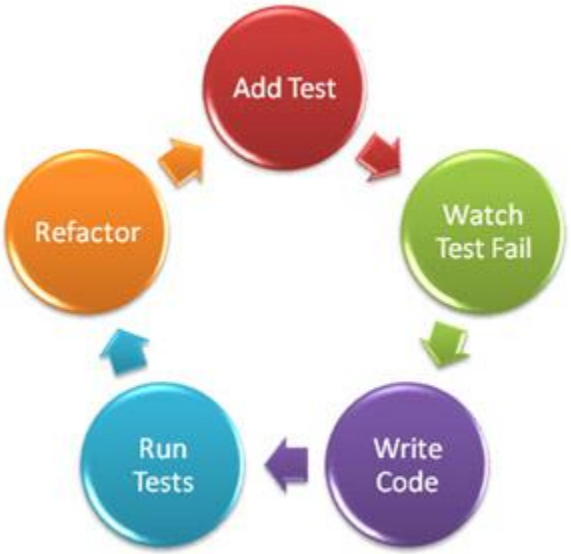
Write tests first

Run the tests (they will probably fail)

Write some code

Rerun the tests

Debug until the tests pass



- Relatively little untested code at any one time so bugs are likely to be in the most recent code

TDD provides useful feedback

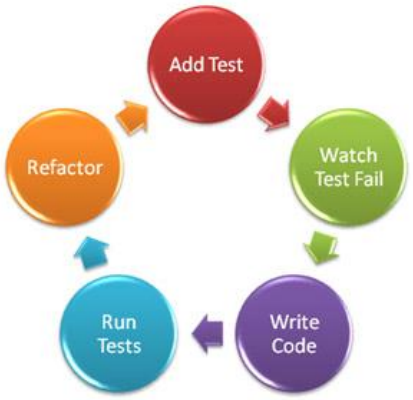
Programmers get feedback when their tests pass

Programmers get feedback when their tests fail

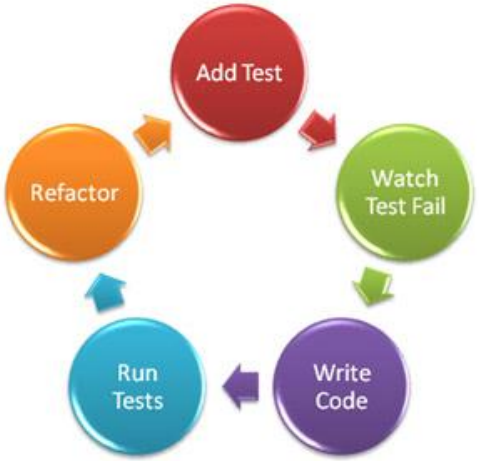
Customers get feedback when the tests pass or fail

- Provides insights on how the developers are doing

Provides frequent metrics for management



Pace



Write a few tests

Write a few lines of code

Don't write too many tests at once

Don't write too much code at once because it's harder to debug until you know it works properly

xUnit

Framework for unit tests, e.g. jUnit, PyUnit(unittest)

Tests are written as class methods

- Developers appreciate that tests are code

Test code is separate from production code

There are similar xUnit frameworks and tools for many other programming languages

- CUnit
- CppUnit
- csUnit
- ...

Example Test Fixture


```
public class MoneyTest extends TestCase {
 //Create a class to test adding Swiss Francs
 public void testSimpleAdd() {
   Money m12CHF= new Money(12, "CHF");  // fixture
   Money m14CHF= new Money(14, "CHF");
   Money expected= new Money(26, "CHF");

   Money result= m12CHF.add(m14CHF);    // exercise

   Assert.assertTrue(expected.equals(result));
  }
}
```

Source: Gamma1998

Architecture of a test

1. **fixture** – creates objects and context for test
2. **exercise** – invokes methods under test
3. **result** – asserts equality (or something else) about the results of the exercise

   Assert equal, not equal, close, membership, etc.

Combining tests in a suite

Collect all the tests for a class in a test suite:

1. Create a new instance of TestSuite
2. Use the addTest method to include the tests you have written

Simple Python Unittest recipe

1. `import unittest`
2. Derive a class `fooTest` from `unittest.TestCase` for the class/feature `foo` being tested (the test class name is arbitrary)
3. Define a set of methods inside `fooTest` for each test case

   Each test case includes calls to `unittest.TestCase.assert*()` methods

   Method name should begin with '`test_`'
4. Call `unittest.main()`

   `unittest.main()` automatically invokes all of the methods in all classes derived from `unittest.TestCase`
5. Debug and fix bugs in test cases and code
6. Repeat until all tests pass

Python unittest Assert Methods

Method	Checks
assertEqual(a, b, msg=None)	a == b
assertNotEqual(a, b, msg=None)	a != b
assertAlmostEqual(a, b, places=7,msg=None)	round(a-b, places) == 0
assertNotAlmostEqual(a, b, places=7,msg=None)	round(a-b, places) != 0
assertTrue(v, msg=None)	bool(v) is True
assertFalse(v, msg=None)	bool(v) is False
assertIs(a, b, msg=None)	a is b
assertIsNot(a, b, msg=None)	a is not b
assertIsNone(v, msg=None)	v is None
assertIsNotNone(v, msg=None)	v is not None
assertIn(a, b, msg=None)	a in b
assertNotIn(a, b, msg=None)	a not in b
assertIsInstance(a, b, msg=None)	isinstance(a, b)
assertNotIsInstance(a, b, msg=None)	not isinstance(a, b)
assertRaises(Exception, function, [function args])	Exception is raised

Source: https://docs.python.org/dev/library/unittest.html

From TDD to Behavior-Driven Development (BDD)

Customers and some developers have trouble defining tests:

- Where to start?
- What to test?
- What to call the tests?

How can we improve the test process to include the customer?

- Change the syntax and simplify the process!

Simplify testing

Replace source code with natural language descriptions

- Eliminate the word "Test"

Change from:

```
public class CustomerLookupTest extends TestCase {
    testFindsCustomerById() { ... }
    testFailsForDuplicateCustomers() { ... }
    ...
}
```

To:

CustomerLookup
- finds customer by id
- fails for duplicate customers
- ...

User Story templates don't map easily to tests

As a [X]

I want [Y]

so that [Z]

As a customer,

I want to withdraw cash from an ATM,

so that I don't need to wait in line at the bank.

Behavior templates aid testing

Given some initial context (the givens),

When an event occurs,

Then ensure some outcomes.

Given the account is in credit

And the card is valid

And the dispenser contains cash

When the customer requests cash

Then ensure the account is debited

And ensure cash is dispensed

And ensure the card is returned

Pre-conditions

Event

Post-conditions

Evolving User Stories to Behavior Stories

Behavior stories are still easy for customers to write and understand

Describe what needs to happen

BUT…

Behavior Stories are easier to map automatically from user descriptions to executable test cases

BDD Tools

JBehave: JUnit for customers

RBehave: JBehave in Ruby

Behave: Behave for Python

RSpec: Evolution of RBehave

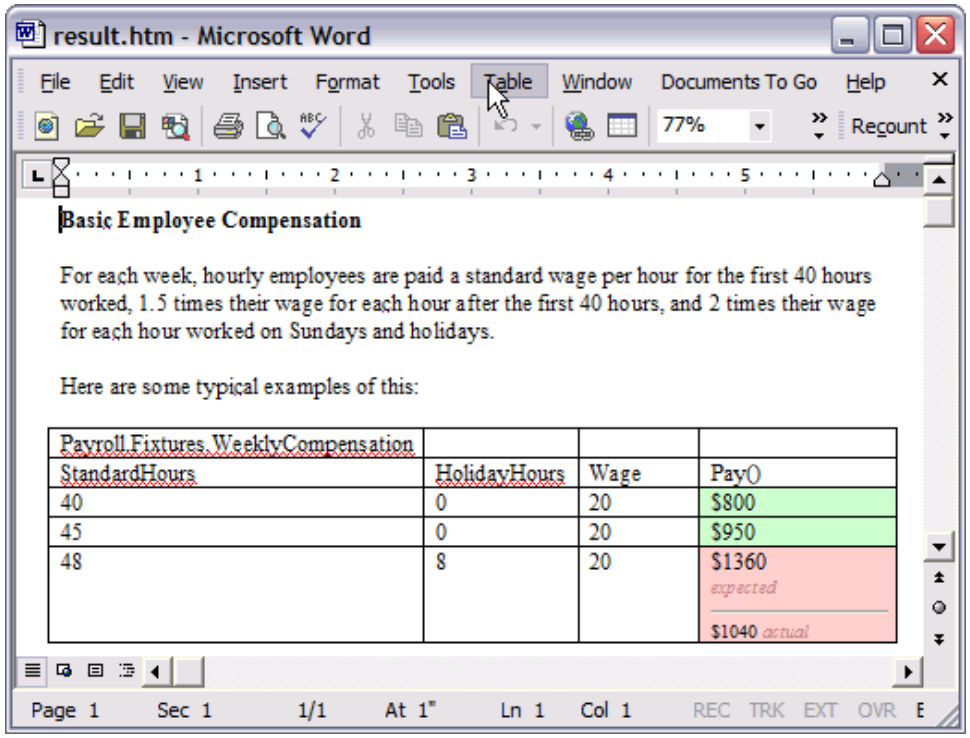
Cucumber: UI specs in natural language for Ruby

... similar tools for Python, C, C++, Delphi, PHP, .Net

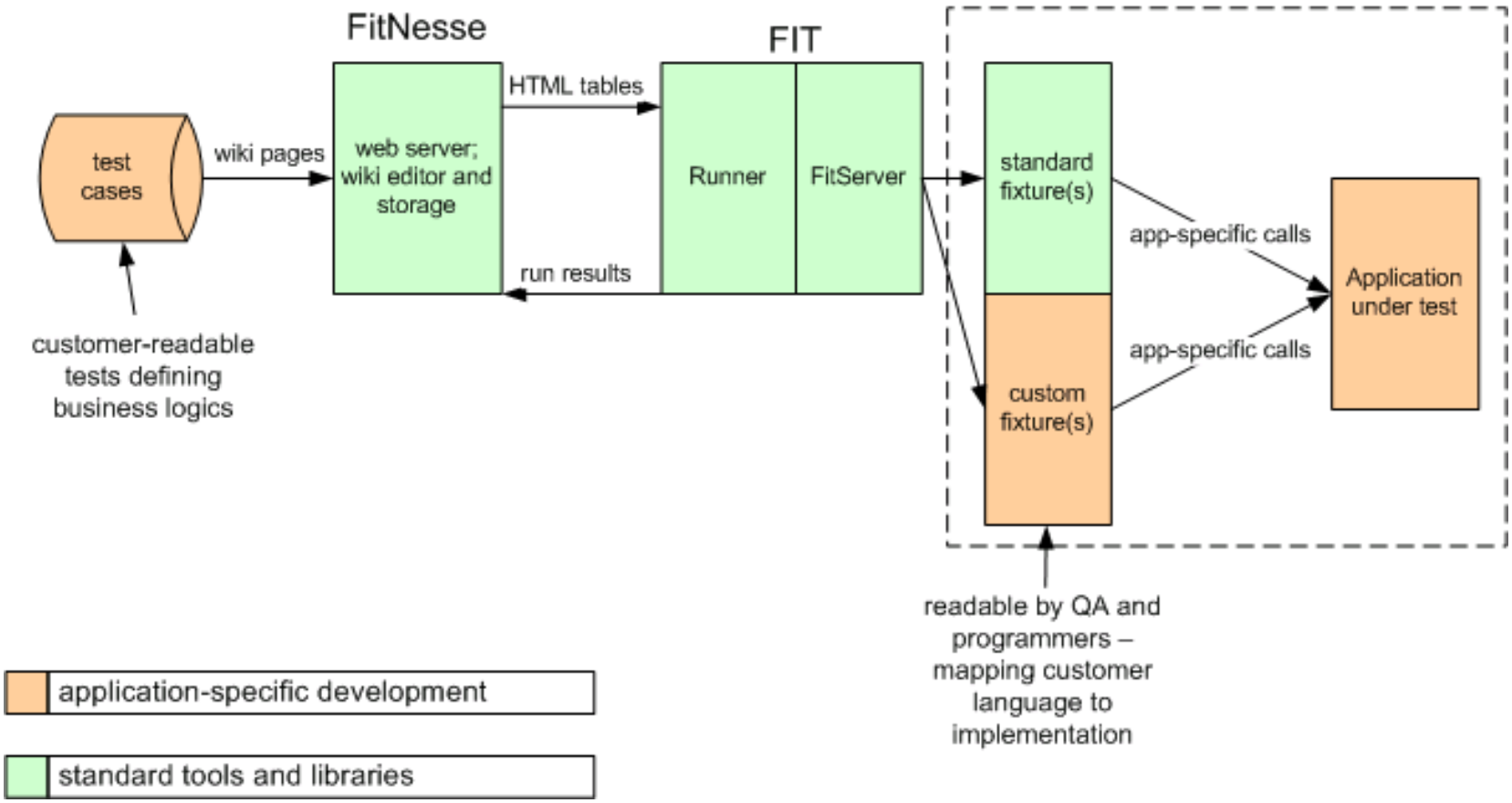
Framework for Integrated Test (FIT) for Acceptance Testing

FitNesse

- Developed by Ward Cunningham as an extension of xUnit framework
- Encourages customer participation via simple tables



FIT + Wiki + Web Server = FitNesse



Why use a Wiki?

Lower the barrier to customer participation

- Easier from user to access with web browser
- Ward Cunningham invented Wikis so, why not them here?

Easy to keep up-to-date

Fixture: Connection between test system and application

When "Test" button is pushed, a fixture is called to process the table

The fixture delegates to underlying application code

- Map the user's test to the relevant application code

Fixture code is like xUnit TestCase code

- Extends base class
- May create objects for multiple tests

Common table formats for tests

ColumnFixture

- Each row specifies one input and one output
- One input or output may be a collection of values

RowFixture

- First row is input, remaining rows are output
- Analogous to a query

ActionFixture

- Each row is either an action to perform or a value to check
- Analogous to a state machine

ColumnFixture: Input/Output

eg.Division		
Input	Input	Output
numerator	denominator	quotient?
10	2	5
12.6	3	4.2
100	4	25

RowFixture/Query

fitnesse.fixtures.PrimeNumberRowFixture	
prime	Query
2	Output
3	Output
5	Output
7	Output

ActionFixture/State Machine

Useful for specifying tests for User Interfaces

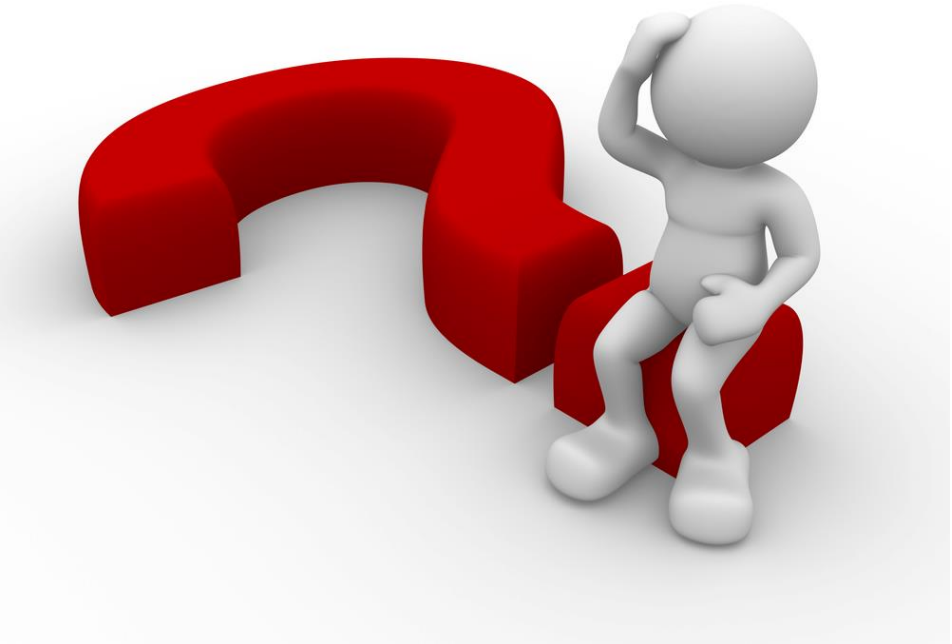
1. Start
2. Check counter
3. User hits 'Press'
4. Check counter

Action Table		
start	fitnesse.fixtures.CountFixture	
check	counter	0
press	count	
check	counter	1

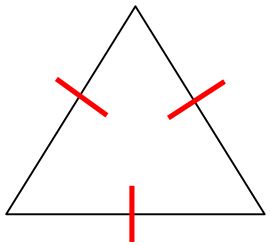
FitNesse Summary

- Goal is to engage customer in testing through an easy to use UI
- Customer writes test specifications in an easy to use domain specific language (DSL)
- Tools convert instructions in DSL to xUnit commands or equivalent

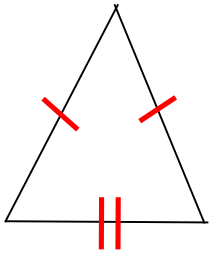
Questions?



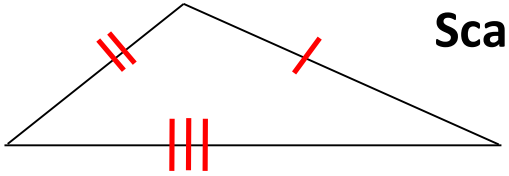
Testing triangle classification



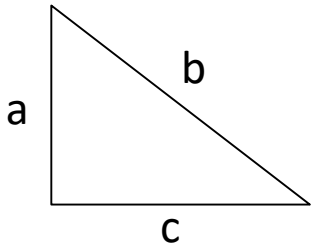
Equilateral: all three sides have the same length



Isosceles: exactly two sides have the same length



Scalene: sides have the three different lengths



Right: $a^2 + b^2 = c^2$