SSW-555: Agile Methods for Software Development

Testing and Continuous Integration

Dr. Richard Ens
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

Software testing

Traditional Methods

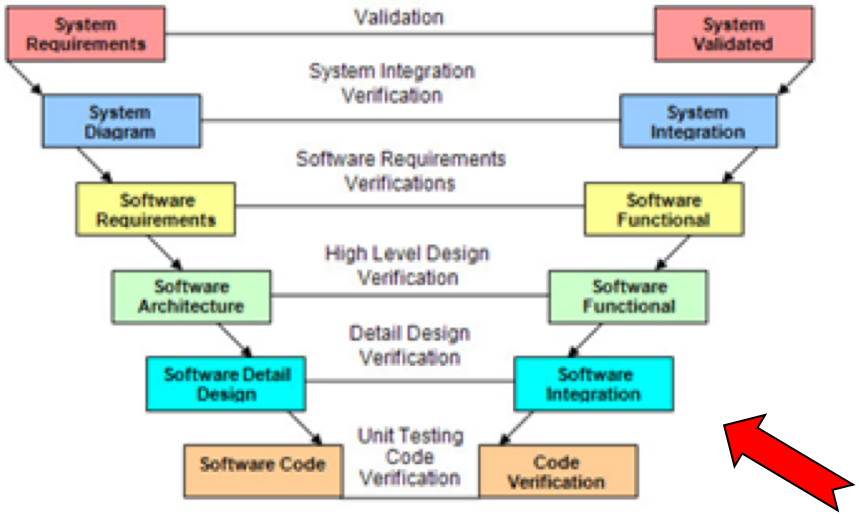
React to quality issues

Agile Methods

Proactively reduce quality issues

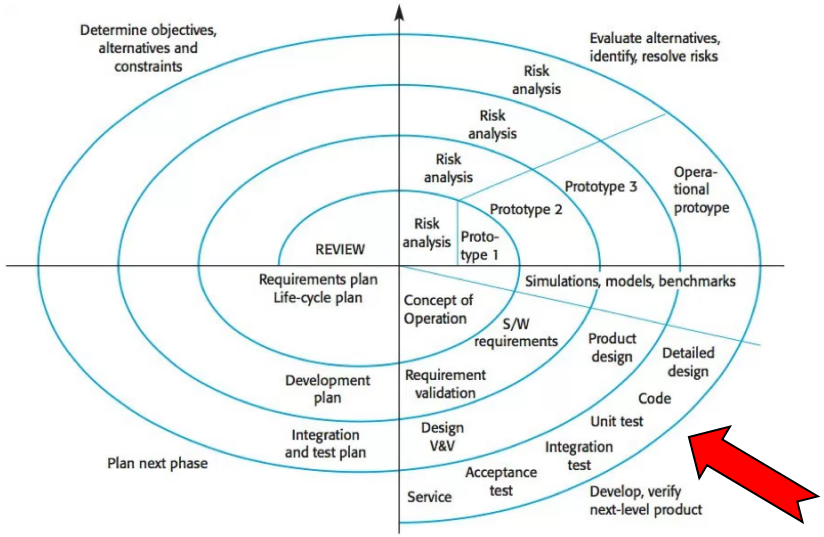
Traditional Software testing

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: Unit Testing

Unit Testing

- Ensures that recently changed unit works correctly

Who?	What?	When?
Developers	New features or bug fixes; Code/branch coverage	After code complete; before integration

Traditional testing: Integration Testing

Integration Testing

- Ensures that components, potentially written by different developers, integrate cleanly and work together

Who?	What?	When?
Developers and/or testers	Test multiple modules from potentially different developers	After unit test is complete on relevant modules

Traditional testing: Regression Testing

Regression Testing

- Verify that changes haven't introduced new problems in already tested code

Who?	What?	When?
Testers (Not the developers)	Integrated modules	After changes to code for bug fixes or new features

Traditional testing: System Testing

System Testing

- Ensures that complete system is correct
- May include functionality, performance, stress testing, etc.

Who?	What?	When?
System Test Team (not the developers)	Test complete system; performance and security; stress testing	After all code modules are complete and tested

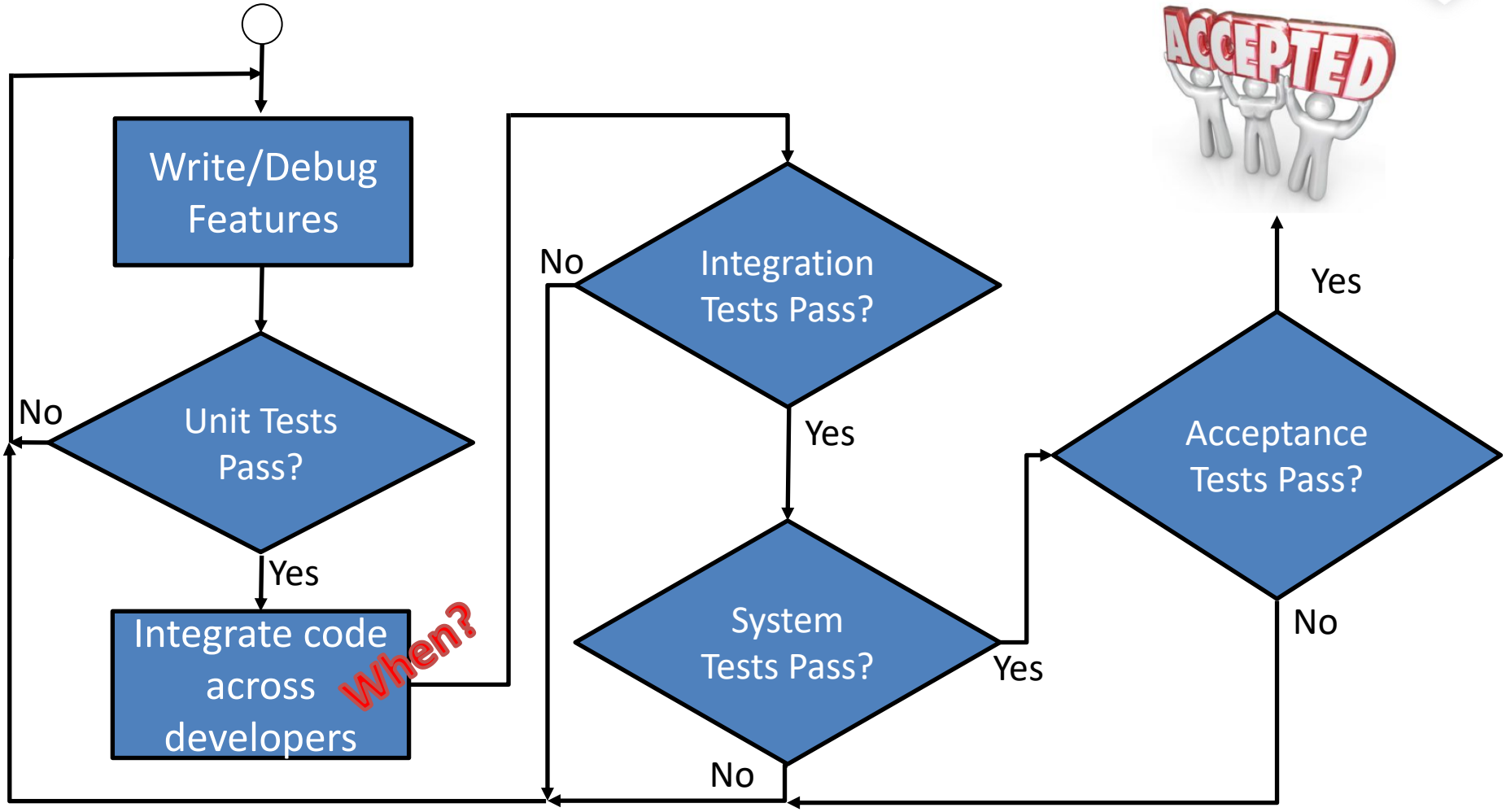
Traditional testing: Acceptance Testing

Acceptance Testing

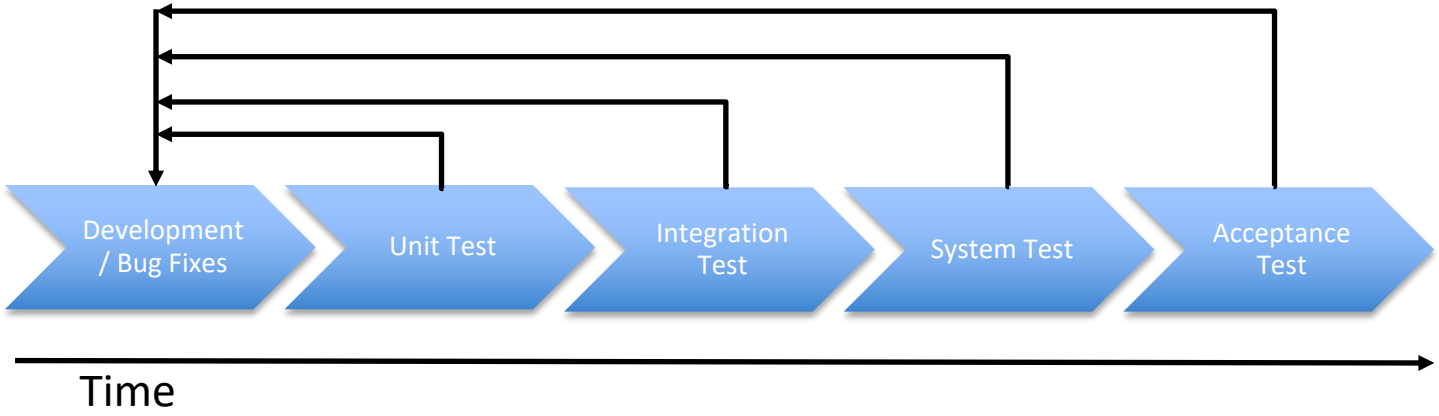
- Customer verifies that the system performs as expected and meets requirements

Who?	What?	When?
Customer test engineers	Complete system based on requirements	After System Test, frequently before final payment

Traditional testing flow



Traditional testing gates



Testing is blocked until development is "complete"

Testing is performed by a separate test team

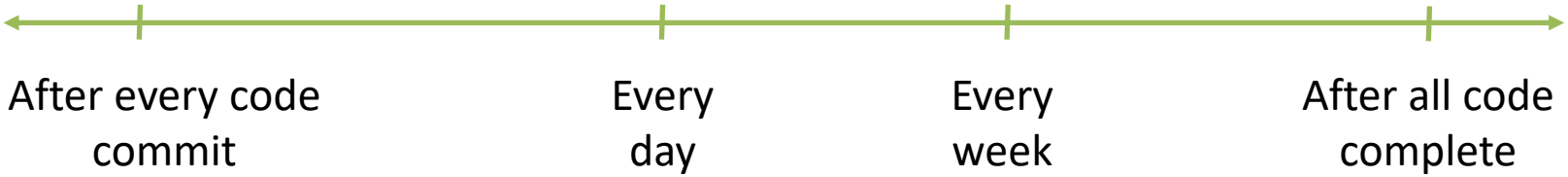
Developers might not be trusted to test their own code

Testing schedule may be compressed if development schedule slips

When to integrate?

Integration combines code from multiple developers

How frequently? Continuously? Periodically?



Agile Methods

Traditional Methods

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…

Agile testing: Unit Testing

Unit Testing

- Ensures that recently completed unit is correct

Who?	What?	When?
Developers using automated testing platforms	New features or bug fixes Code/branch coverage	Tests written before the code is written. Run tests while developing code

Agile testing: Integration Testing

Integration Testing

- Ensures that components, potentially written by different developers, integrate cleanly and work together

Who?	What?	When?
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

Agile testing: Regression Testing

Regression Testing

- Verify that changes haven't introduced new problems in already tested code

Who?	What?	When?
Developers using automated test platforms with tests written before code is written	Entire system or selected components	After changes to code for bug fixes or new features

Agile testing: System Testing

System Testing

- Ensures that complete system is correct
- May include functionality, performance, stress testing, etc.

Who?	What?	When?
Developers using automated test platforms with tests written during development	Test complete system Performance Security Stress testing	Optional: may be done before product release cycle after several sprints

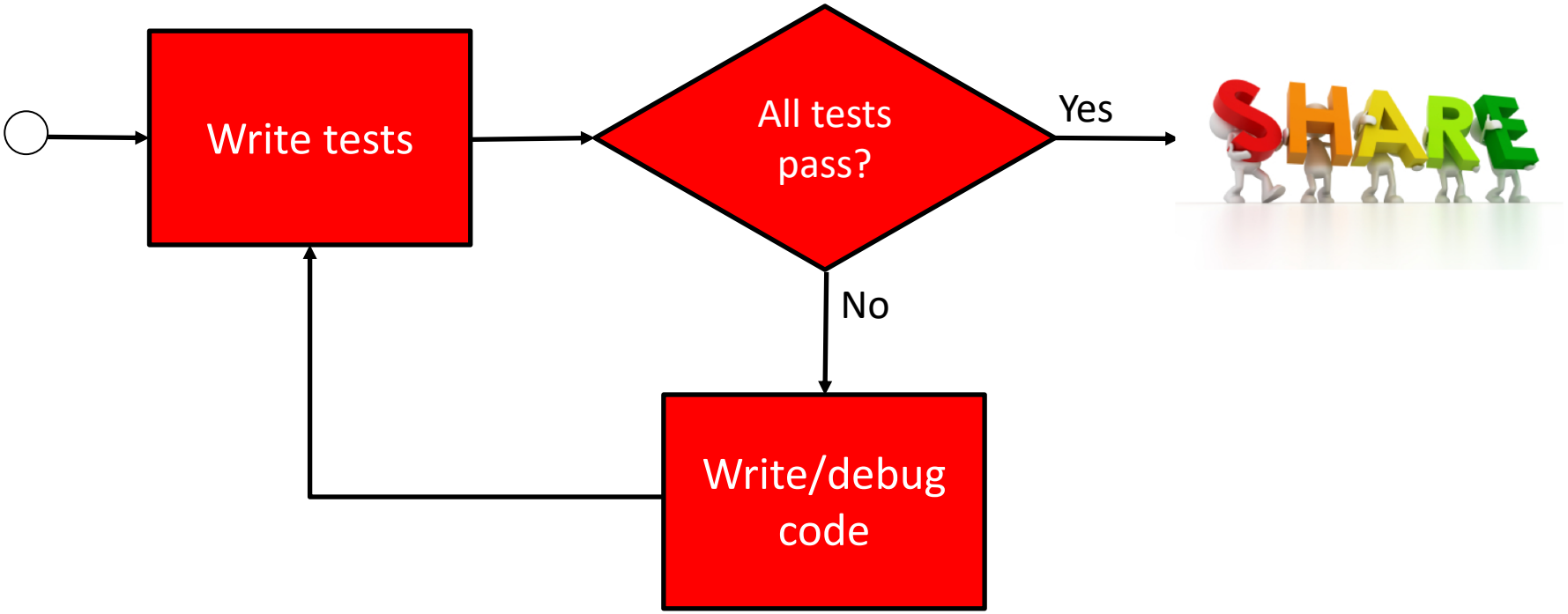
Agile testing: Acceptance Testing

Acceptance Testing

- User verifies that the system performs as expected and meets requirements

Who?	What?	When?
Product Owner, stakeholders	Overall system, including new features from latest sprint	Sprint Review

Agile testing flow: automated and continuous



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?

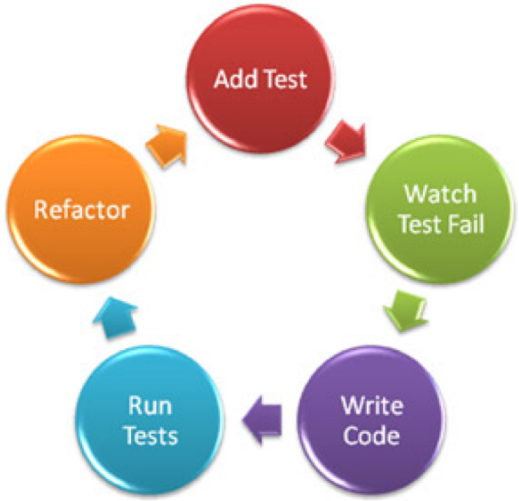
Add bugs as user stories to the product backlog

- 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

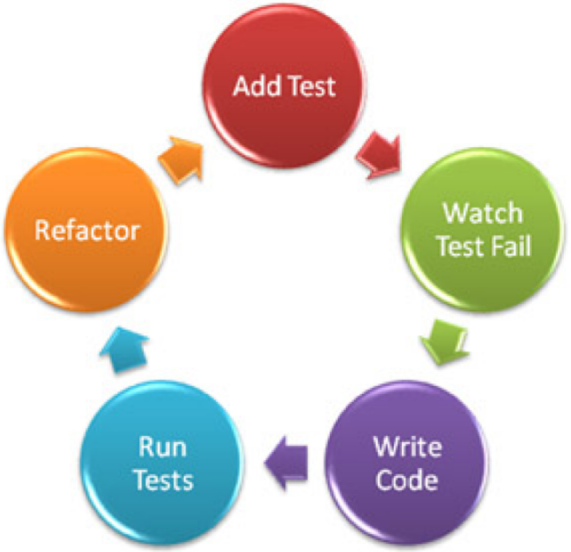
Write tests first

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

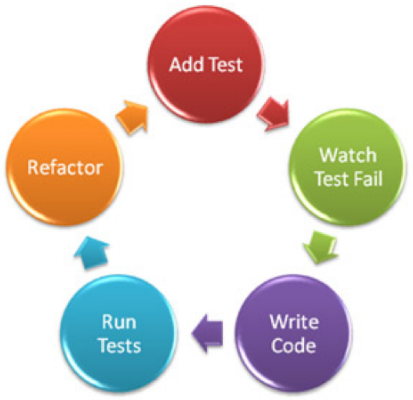
Programmers get feedback when their tests pass

Programmers get feedback when their tests fail

Customers get feedback when the tests pass or fail

- Provides insights on how the developers are doing

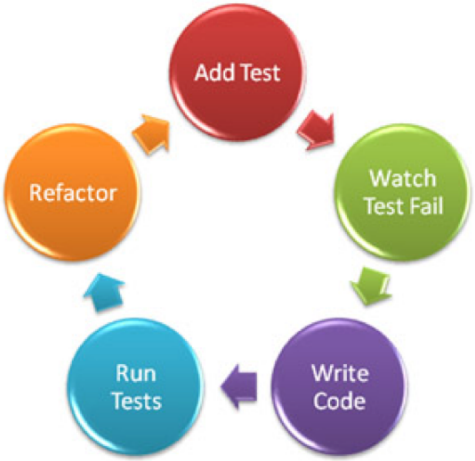
Provides frequent metrics for management



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

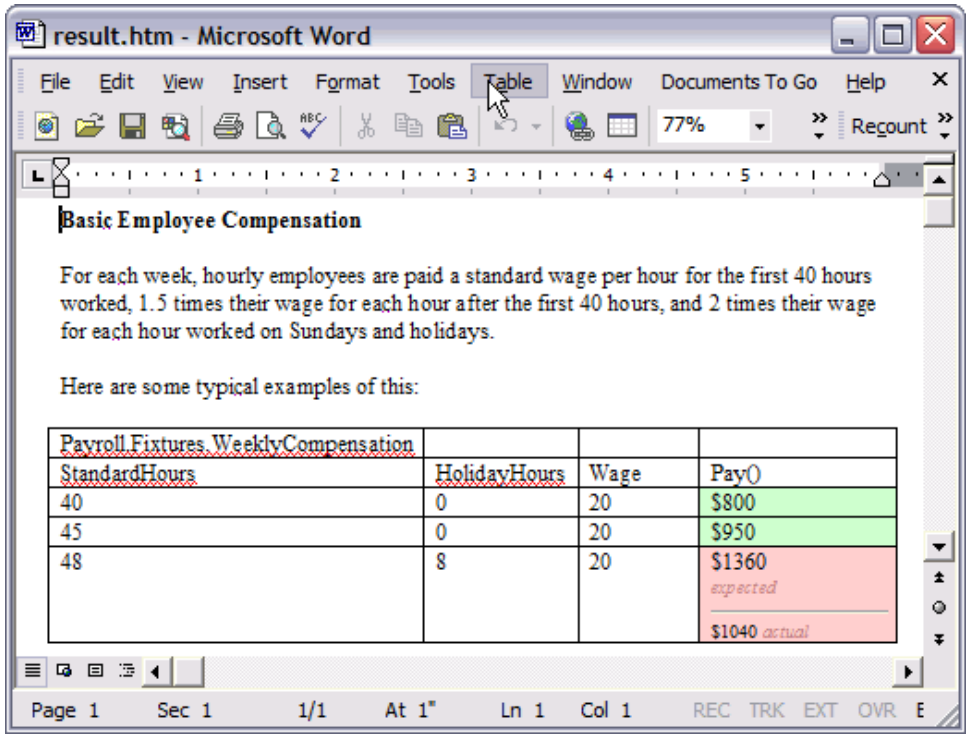
Cucumber: UI specs in natural language for Ruby

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

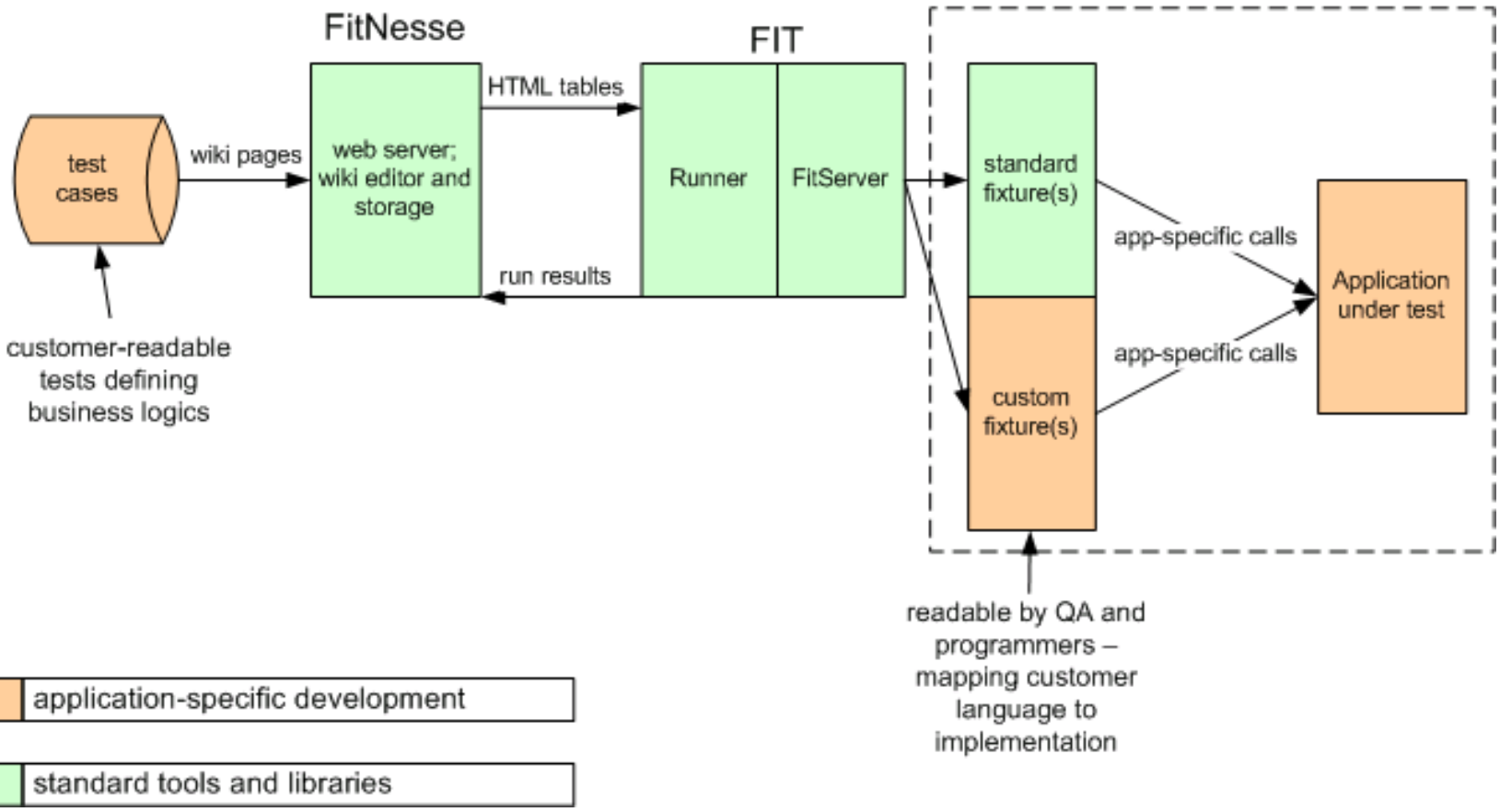
Framework for Integrated Test (FIT) for Acceptance Testing

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'

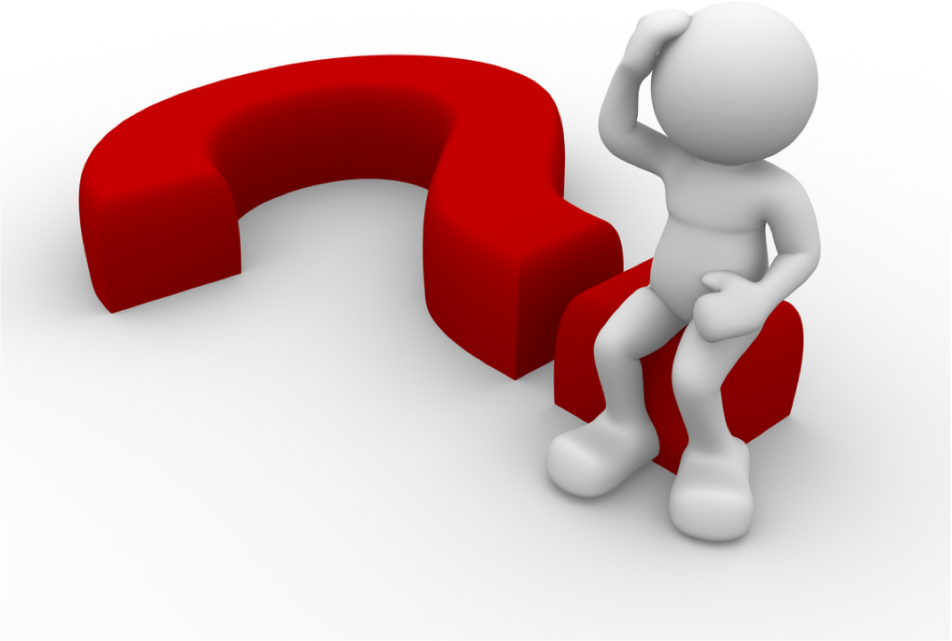
4. Check counter

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

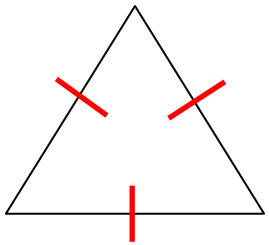
FitNesse Summary

- 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

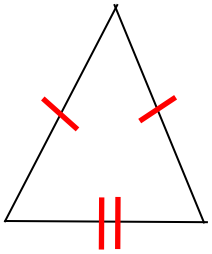
Questions?



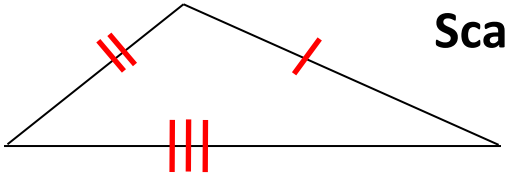
Testing triangle classification



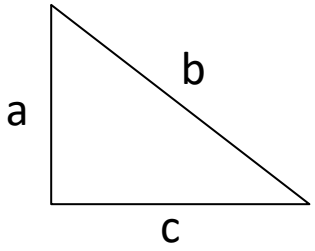
Equilateral: all three sides have the same length



Isosceles: exactly two sides have the same length



Scalene: sides have the three different lengths



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