

# ***Ten Tenets of Testing***

## ***"Key Testing Principles"***

Joseph W. Yoder  
Teams That Innovate  
The Refactory, Inc.

joe@refactory.com

<http://www.refactory.com>

<http://www.teamsthatinnovate.com>

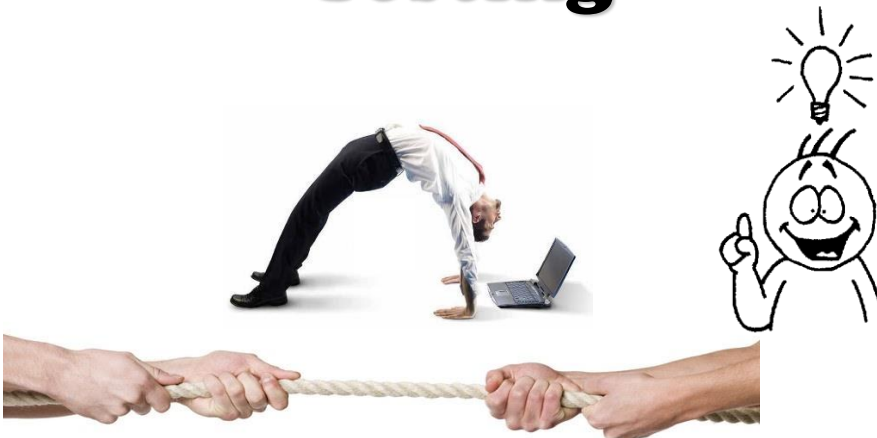


Refactory

© Teams That Innovate  
The Refactory, Inc. & Joseph W. Yoder  
All Rights Reserved.



## **Testing**



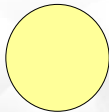
# Understanding Tests

**T**

**Test Target** → the thing we are trying to test



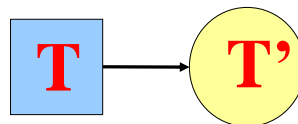
**Action** → changes environment or the Test Target



**Assertion** → compares expected vs observable outcome of the action on the Test Target



**Test** → a sequence of at least one action and one assertion



## Good Test Outline

1. Set up
2. Declare the expected results
3. Exercise the test
4. Get the actual results
5. Assert that the actual results match the expected results
6. Teardown

## FIRE

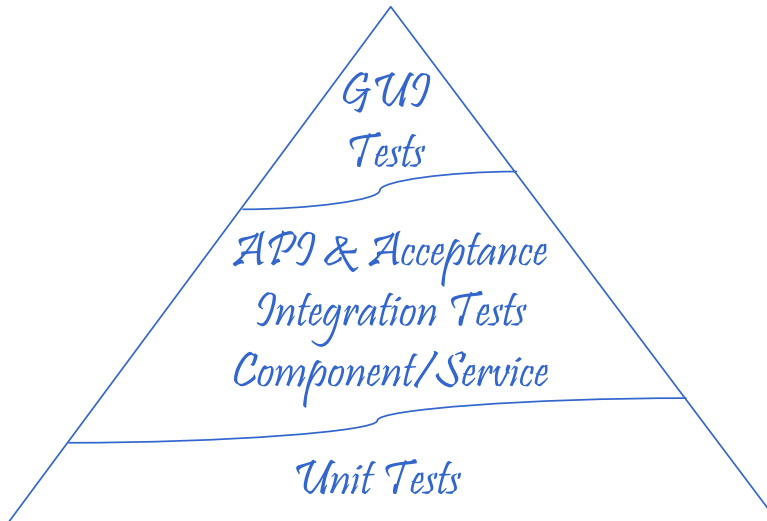
---

Good tests are:

- Fast
- Informative
- Reliable
- Exhaustive



# Testing Pyramid



## Types of Tests

Unit Tests – Tests classes and components

Integration Tests – Tests code integration

Smoke Tests – Quick tests of core functionality

Performance Tests – Test system under load

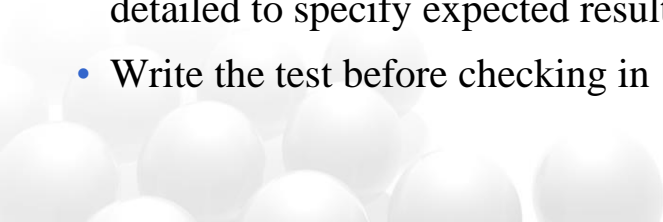
Regression Tests – Tests it is still working

Acceptance Tests – Requirements testing

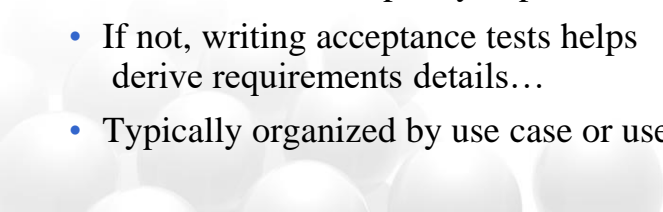
System Tests – Test all parts together

*Who Tests, Who Develops Them,  
Where and How (can we automate)*

## Unit tests

- Tests public interface to functionality
  - Functional test of system such as a class or component (functions within a service)
  - Compare *expected* against *actual* results
  - Assumes requirements are sufficiently detailed to specify expected results!
  - Write the test before checking in
- 

## Acceptance tests

- Mostly functional tests of system delivered by the development team (internal or external)
  - Sometimes testing critical system characteristics (non-functional qualities)
  - Describes specific tests and expected results
  - Requirements must be known in enough details in order to specify expected results!
  - If not, writing acceptance tests helps derive requirements details...
  - Typically organized by use case or user story
- 

## Acceptance tests



- User Acceptance Tests
  - assess whether the Product is working for the user, and specifically correctly for the usage scenario
- Business Acceptance Tests
  - assess whether the Product meets the business goals and purposes or not
- Contract Acceptance Tests
  - tests the contract for when the Product goes live
- Regulation/Compliance Acceptance Tests
  - assess whether the Product violates the rules and regulations of government or country

## Acceptance Tests

Acceptance Criteria can be written in different formats. There are two most common ones, and the third option is to devise your own format:

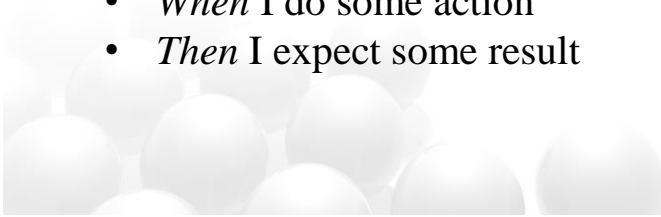
- scenario-oriented (Given/When/Then)
- rule-oriented (checklist)
- custom formats (matching invariants)

# Acceptance Tests

## Scenario-oriented acceptance criteria

Scenario-oriented format of writing AC is known as the *Given/When/Then* (GWT) type:

- *Given* some precondition
- *When* I do some action
- *Then* I expect some result



# Acceptance Tests

**User story:** *As a user, I want to be able to request the cash from my account in ATM so that I will be able to receive the money from my account quickly and in different places.*

## Scenario: Account Overdrawn

**Given:** that the account is overdrawn

**And:** the card is valid

**When:** the customer requests the cash

**Then:** ensure the rejection message is displayed

**And:** ensure cash isn't dispensed



## Agile Acceptance Tests

### Goals

- Baseline confidence that the software works as promised
- Focus on functionality and important qualities such as: performance, load, security...
- Provide immediate feedback
- Tests for story completion in a current iteration of dev

### Not Intended to be:

- Exhaustive, impossible to test 100% of everything
- Test every scenario, UI edit/detail
- Replacement for other testing: unit tests, integration tests, regression tests...

## You Can't Write Acceptance Tests Forever...

- Tests should be written based on business value
- Identify meaningful path combinations through a use case
- Develop test cases for each important path case

Likelihood/ Importance	Paths	
High/High	Basic Flow "Happy Path"	
High/High	Basic Flow	Variation 1
High/Medium	Basic Flow	Variation 2
High/High	Basic Flow	Exception 1
Low/Very Low	Basic Flow	Exception 2



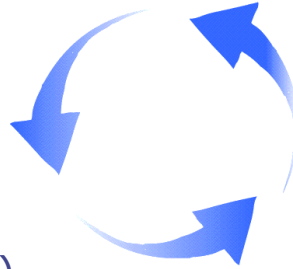
## Good Testing Values

Work in short cycles of testing and coding

Unit Tests written along with other tests (acceptance, ...)

It isn't enough to write tests: you have to run them frequently (many times a day)

Developers get immediate feedback on how they're doing...publish scores and keep track of what is happening (visible)



## Ten Tenets of Testing

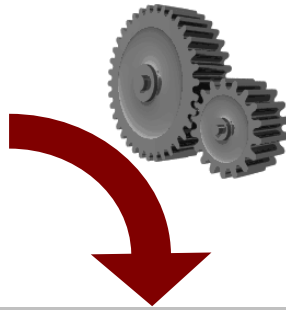


# Ten Tenets of Testing

0. Validate your Tests!!!
1. Test single complete scenarios
2. Do not create dependencies between tests
3. Only verify a single thing in each assertion
4. Respect service encapsulation
5. Test limit values and boundaries
6. Test expected exceptional scenarios
7. Test interactions with other services
8. When you find a bug, write a test to show it
9. Do not duplicate application logic in tests
10. Keep your test code clean



RemoteExecutor
+connect ()
+sendExecution ()
+getExecutionStatus ()
+getExecutionResults ()
+getConnectionStatus ()
+disconnect ()



## Automatic Test Generated Code

```
public void testConnect() {}
public void testSendExecution() {}
public void testGetExecutionStatus() {}
public void testGetExecutionResults() {}
public void testGetConnectionStatus() {}
public void testDisconnect() {}
```

## No! Stop and Think...

RemoteExecutor
+ connect ()
+ sendExecution ()
+ getExecutionStatus ()
+ getExecutionResults ()
+ getConnectionStatus ()
+ disconnect ()

**We should test  
successful  
and failed  
connections**

**And scenarios  
with different  
execution status**

```
@Test
public void statusOfActiveTask() {
    ex.connect(ip);
    ex.sendExecution("TASK");
    String status =
        ex.getExecutionStatus("TASK");
    assertEquals(ACTIVE, status);
}
```

**A test should exercise usage scenarios of the tested service, which can invoke several functions**

## **Don't Test Too Much, and Don't Test Too Little...**

Test single complete scenarios

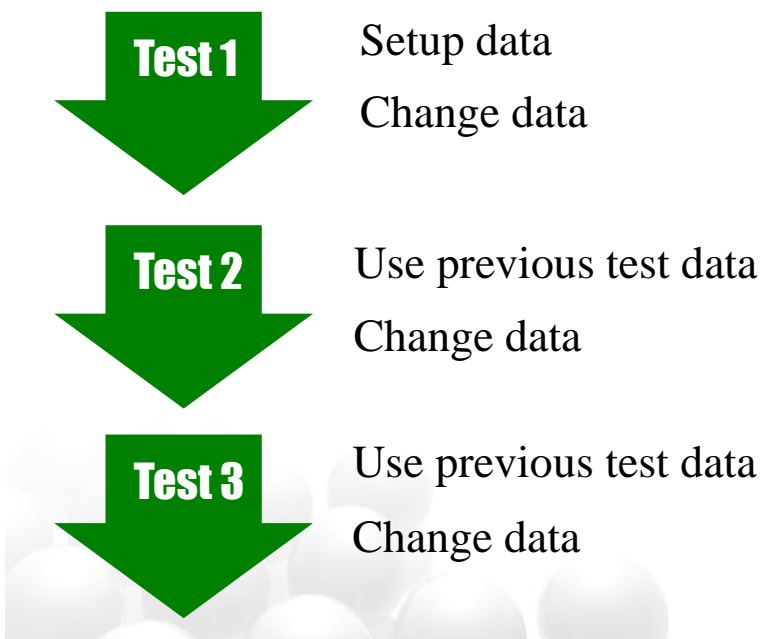
Test only one scenario in a test

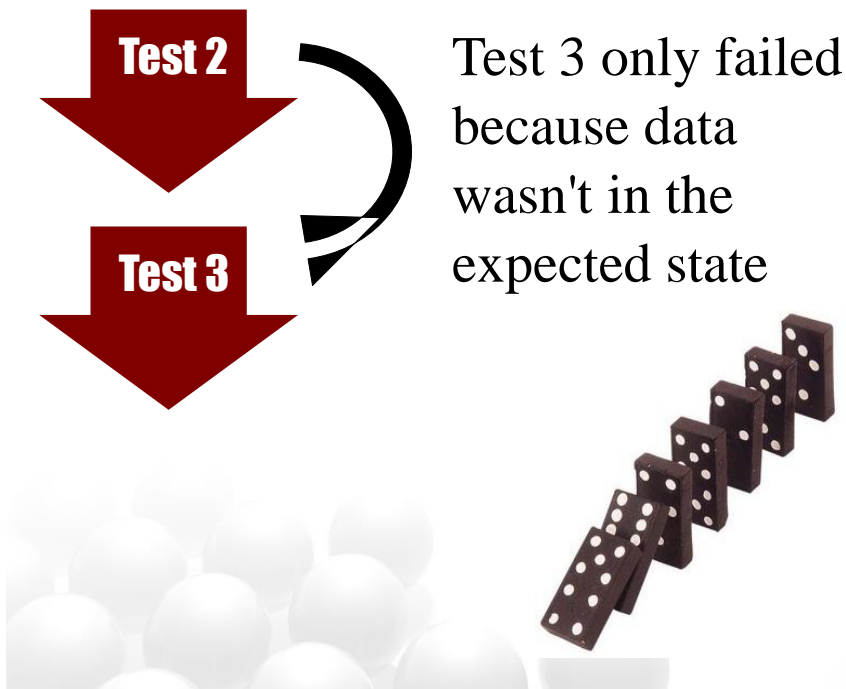
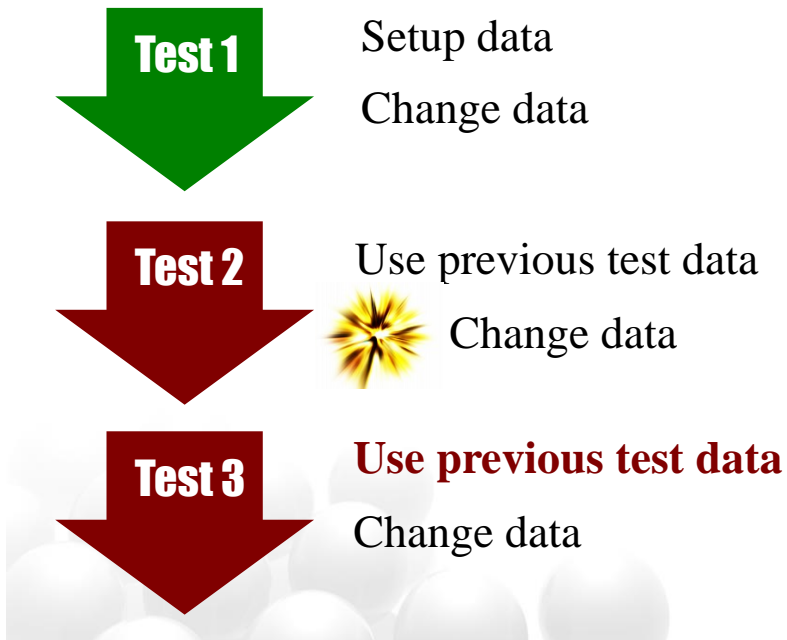
Do not just test getters or setters

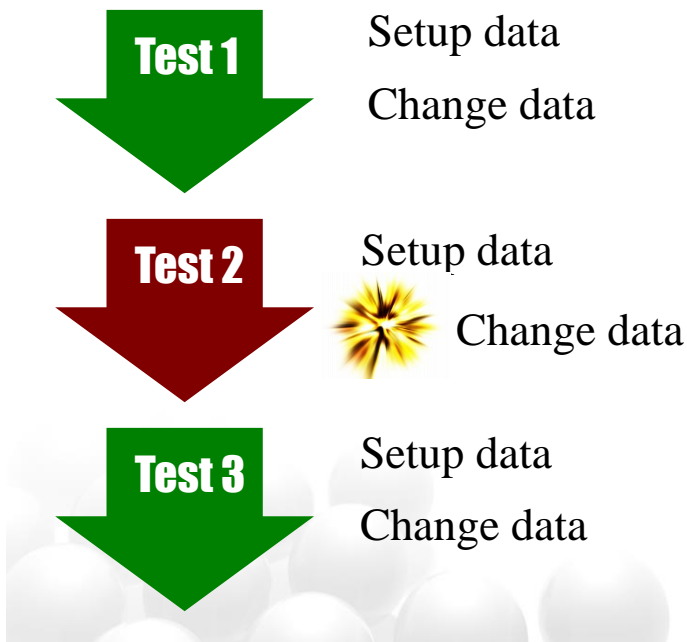
*"unless they have side effects"*

Do not test simple support method calls that are not part of real scenarios, real scenarios should test these

Most important to Test Real Business Usage Scenarios







**With independent tests  
it is easier to identify  
which verifications  
actually failed**

# 3 Verify a Single Thing in Each Assertion



```
assertTrue(  
    student.getFinalGrade() == 6.0  
    && !student.isApproved()  
    && student.status().equals(LOCKED) );
```



**If the assertion fails, where is the problem?**

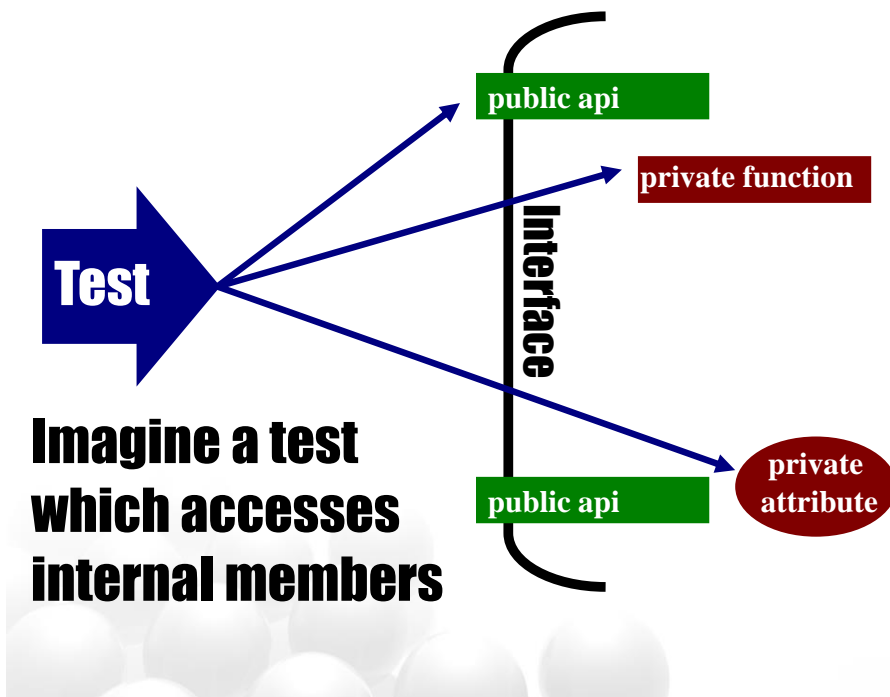
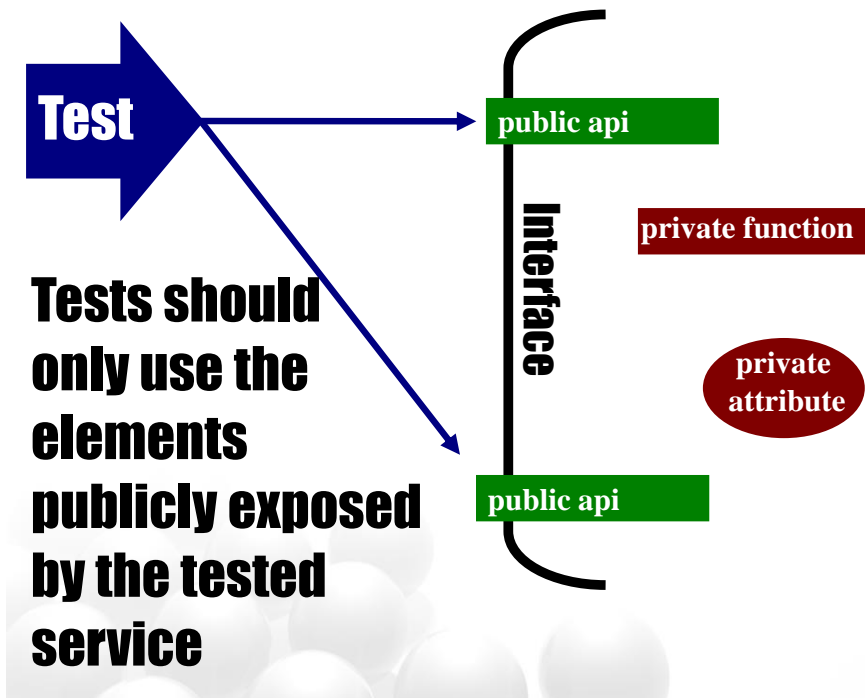


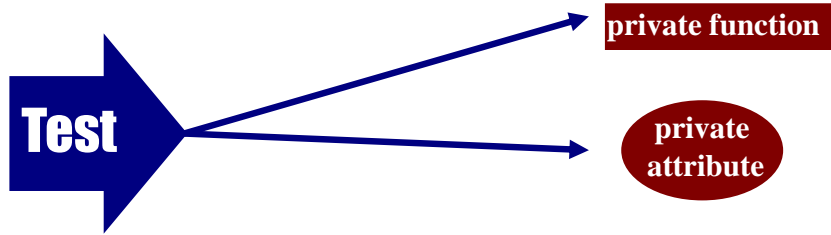
```
assertEquals(6.0,  
             student.getFinalGrade());  
assertFalse(student.isApproved());  
assertEquals(LOCKED, student.status());
```

**The code is more  
readable and, if the test  
fails, the problem is  
more easily located!**

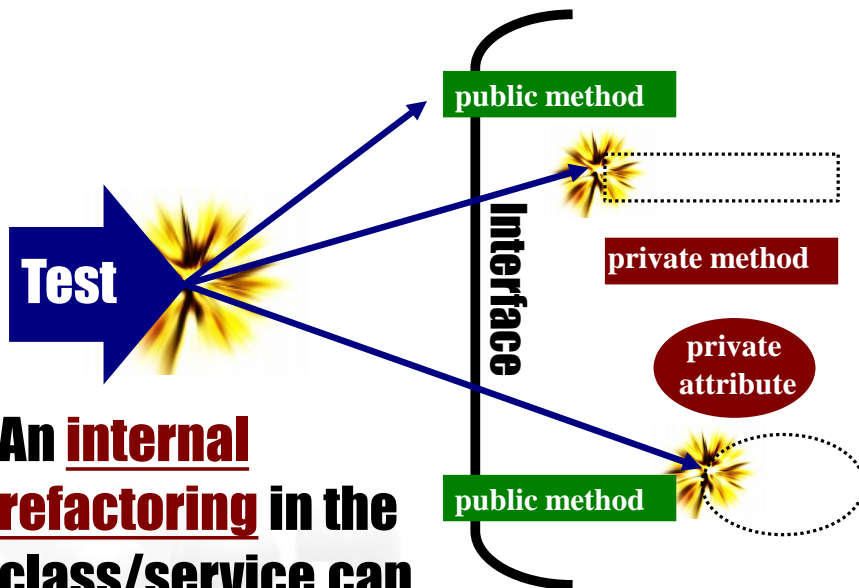


**4** Respect  
Class/Service  
Encapsulation

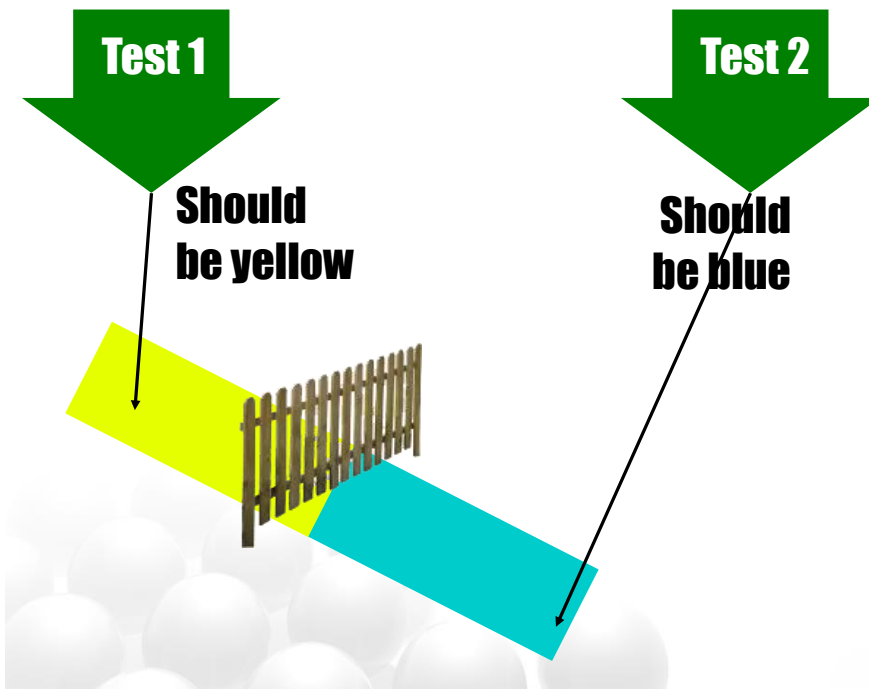




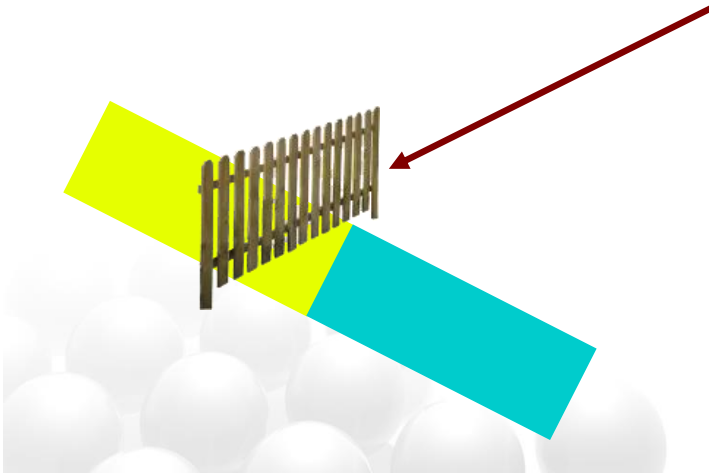
**Inappropriate Intimacy**



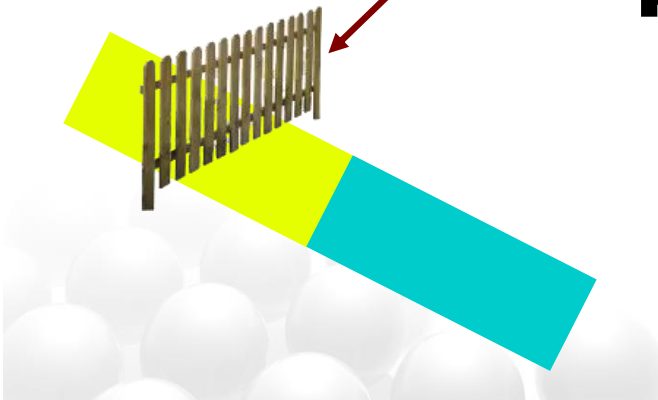
**An internal refactoring in the class/service can break the test!**

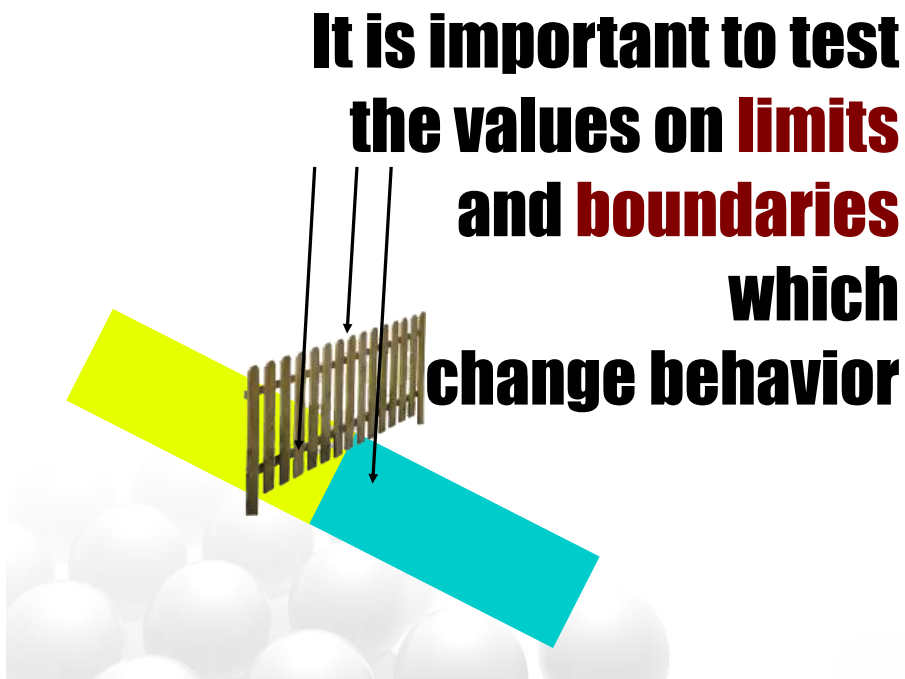


**But what  
about **here**?**

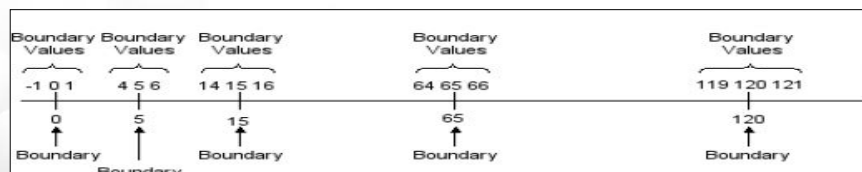
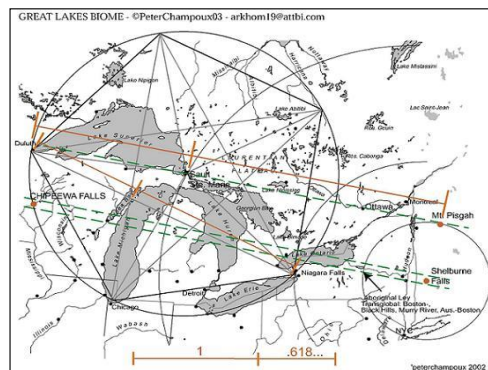


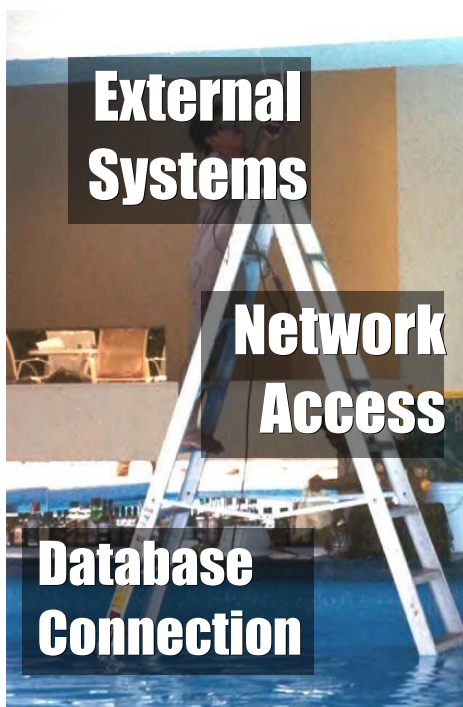
**Is the **boundary** in  
the right  
place?**





## Don't Test All Boundaries





**Sometimes  
you know that  
some things  
can go really  
wrong**

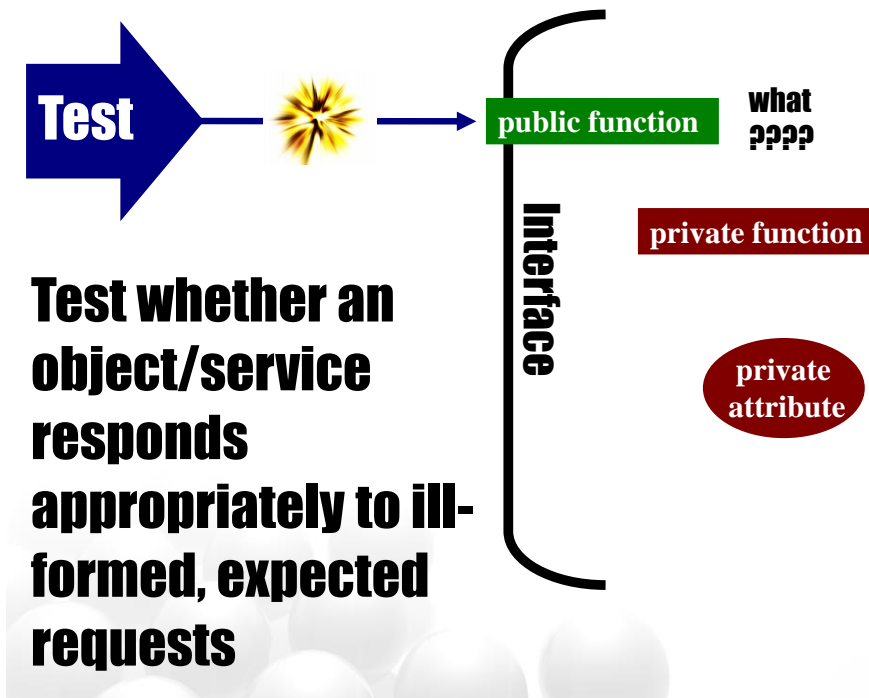




**Clients of your services  
can also do stupid things!**



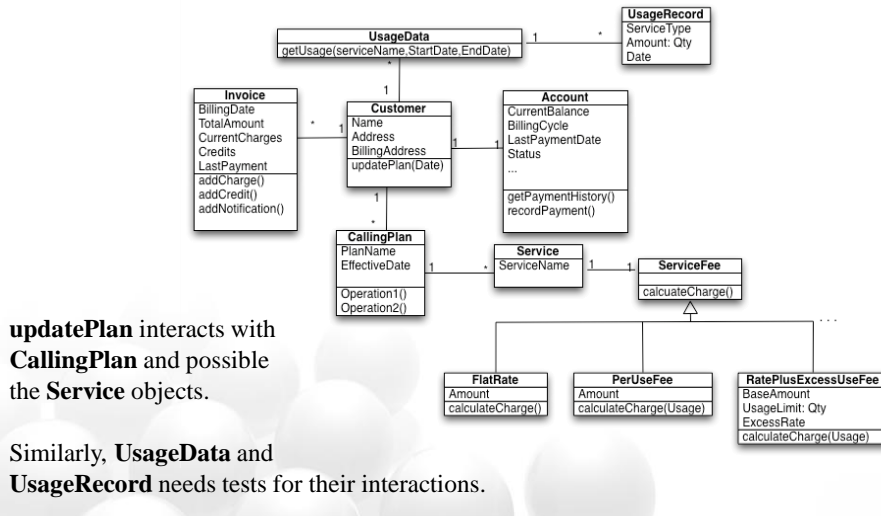




# 7 Test Interactions with other Objects or Services



# Test Service/Object Interactions



# 8 When You Find a Bug, Write a Test to Show It!



## Why Write a Test To Show a Bug?

Demonstrates that it is repeatable.

As more code is written, old problems that were “fixed” can become “broken” again.

A test that validates a bug ensures against reintroducing the problem, i.e. not losing money again

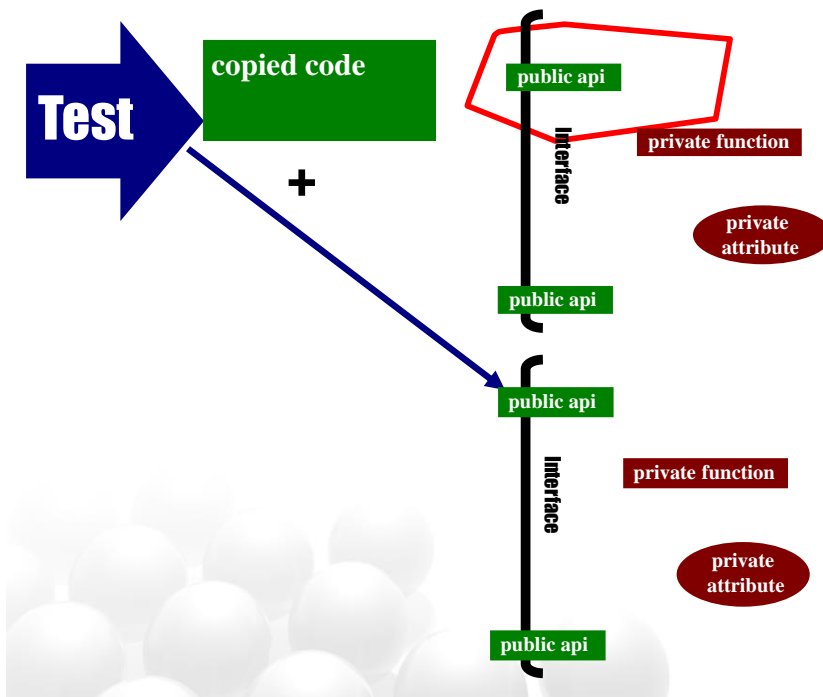


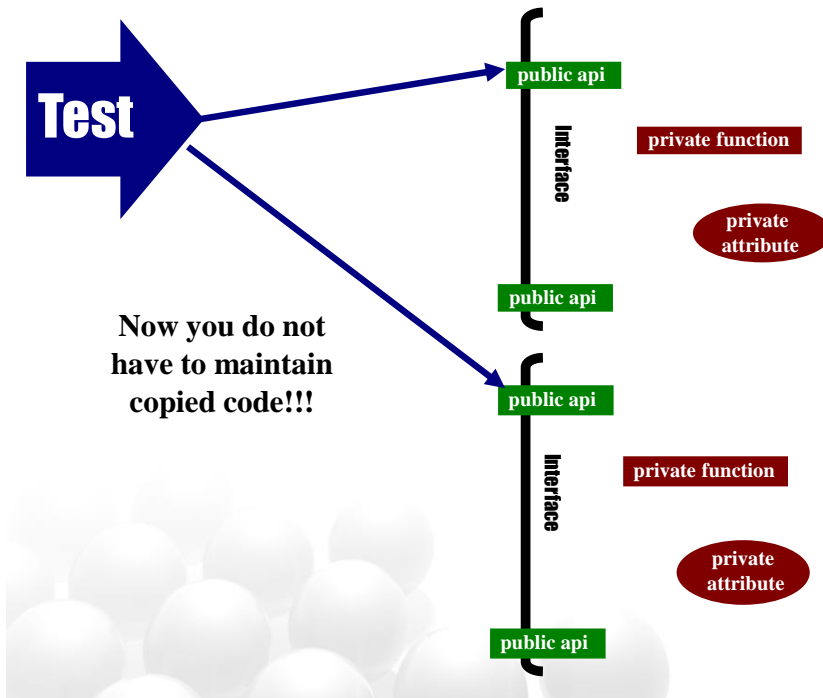
And if my test  
need logic  
present in other  
classes/services?



DRY Principle

**Do not  
duplicate!**

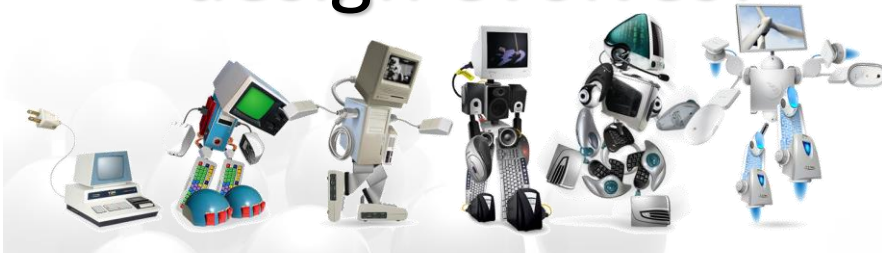


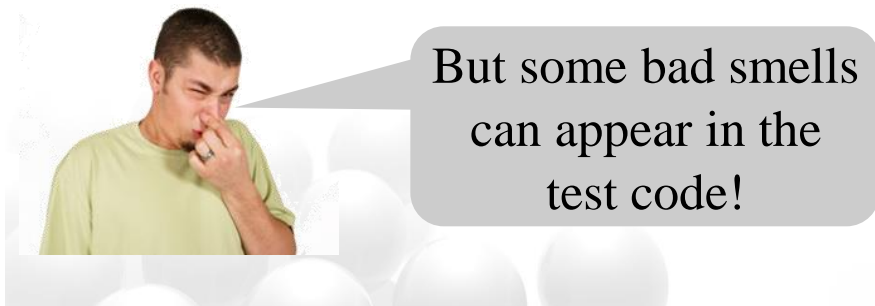
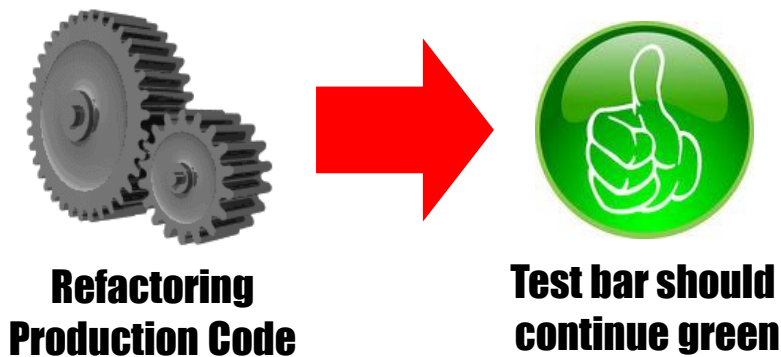


# Tests need Refactoring



Test code design  
evolves, just like  
production code  
design evolves!





## Ten Tenets of Testing

0. Validate your Tests!!!
1. Test single complete scenarios
2. Do not create dependencies between tests
3. Only verify a single thing in each assertion
4. Respect service/class encapsulation
5. Test limit values and boundaries
6. Test expected exceptional scenarios
7. Test interactions with other objects/services
8. When you find a bug, write a test to show it
9. Do not duplicate application logic in tests
10. Keep your test code clean



## Services (threads) can be hard to Test

**Limit** threads as much as possible non-threaded code is easier to test

*When we do have to create threads, consider the following:*

1. Decouple the logic that orchestrate the threads from the ones with the business logic. Then, mock the threads forcing some synchronization scenarios. Test the business logic individually wherever possible.
2. The objects that process information usually are created inside methods as local variables (for instance, inside loops) because they receive as parameters for processing local information. To enable mocking in this case, use a factory for the creation of such objects. So, for testing you can introduce a factory which create the mocks.
3. For testing, add a sleep with different times on the mock execution to simulate different processing orders for different testing scenarios.

## Summary: What to Test

- Test significant business scenarios of use, not isolated functions
- Spend time testing the difficult parts:
  - Complex interactions
  - Intricate algorithms
  - Tricky business logic
- Test for required system qualities
  - Performance, scalability, throughput, security
- Test how services respond to normal and exceptional invocations



## Summary: What Not to Test

- Tests should add value, test real business scenarios, not just be an exercise...
- Do not test:
  - setting and getting values alone  
common language things...
  - every boundary condition; only those with significant business value
  - every exception; only those likely to occur or that will cause catastrophic problems.



Obrigado!!!

joe@refactory.com  
Twitter: @metayoda

www.joeyoder.com  
www.refactory.com