Welcome to the Test Refactoring Workshop

Improving the Maintainability of Your Tests

While you are waiting, please download:

Sample Java Test Code: http://india2018exercisejava. xunittraining.com

Sample C# Test Code: http://india2018exercisecsharp. xunittraining.com

These slides: http://india2018exerciseSlides. xunittraining.com

These links are also at the bottom of the session description at https://confengine.com/agile-india-2018/proposal/5757/test-refactoring-workshop-improving-the-maintainability-of-your-tests

Outline

- **Bad Smells in Test Code**
- **Key Success Factors**
- **Terminology**
- **Key Test Refactorings**
- **Other Kinds of Test Smells**

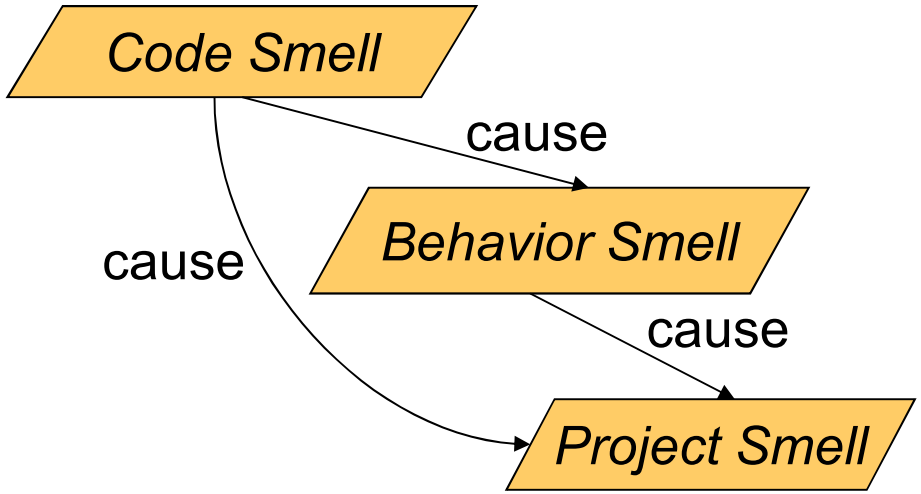
What's a "Smell"?

- **A set of symptoms of an underlying problem in code**
- **Introduced by Martin Fowler in:**
  - Refactoring – Improving the Design of Existing Code
  - Term originally attributed to Kent Beck
- **The "Sniff Test":**
  - A smell should be obvious
  - It should "grab you by the nose"
- **Not necessarily the actual cause**
  - There may be many possible causes for the symptom
  - Some root causes may contribute to several different smells

Note: Past literature often labels the cause as a smell.
e.g. "Sensitive Equality" is really a cause of "Fragile Test"

What's a "Test Smell"?

- **Three common kinds of Test Smells:**
  - Code Smells – Visible Problems in Test Code
  - Behavior Smells – Tests Behaving Badly
  - Project Smells – Testing-related problems visible to a Project Manager
- **Code Smells may be root cause of Behavior and Project Smells**



Code Smells – Obscure Test (1)

- **Eager Test**
  - Testing more than one Test Condition
- **Verbose Test**
  - Using more code than necessary to express the test condition
- **General Fixture**
  - Setting up the (unused) fixture for several Test Conditions in each test
- **Indirect Testing**
  - Interacting with the System Under Test (SUT) via some intermediary object

Code Smells – Obscure Test (2)

- **Hard Coded Test Data**
  - Lots of "Magic Numbers" or Strings used when creating objects.
  - More likely to result in unrepeatable tests
- **Mystery Guest**
  - Lots of "Magic Numbers" or Strings used as keys to database. What do they refer to?
  - "Lopsided" feel to tests (either Setup or Verification of outcome is external to test)
  - Hard to understand what is being tested or what success looks like
- **Other Names:**
  - Verbose Test, Complex Test, Long Test

Code Smells - Conditional Test

- **Conditional Test Logic**
  - Tests containing conditional logic (IF statements or loops)
  - Hard to verify correctness. Does it always test the same thing?
  - What statements are executed?
- **Flexible Test Logic**
  - The test accommodates different results from the SUT in different circumstances
  - Indicates incomplete control of the “test fixture”
- **Complex (Any) Undo Logic**
  - Complex fixture teardown code
  - More likely to leave test environment corrupted

Example of Complex Undo Logic

```
testXxx( ) throws Exception {
    object x, y;
    try {
        x = myApp.insert( "abc");
        y = myApp.insert ("xyz");
    } finally {
        try {
            If (x != null) myApp.delete("abc");
        } finally {
            If (y != null) myApp.delete("xyz");
        }
}
```

This code cannot be tested!

Zen of Testing: "Avoid cleanup complexity"

- **Choose a fixture strategy to avoid complex cleanups.**
- **Cleanup should be non-critical (e.g. use unique keys)**

Code Smells – Test Code Duplication

- The same code appears over and over in the same or any tests
- Changes to SUT require much test code to be modified
  - » **E.g. Adding an argument to a method**

- **Caused by:**
  - Cut & Paste test code reuse, or
  - Reinventing the wheel

Code Smells – Hard-to-Test Code

- **Really a production code smell but test code complexity can be a hint**

Code Smells –Test Logic in Production

- **Another production code smell**

Code Smells –Test Logic in Production

- **Another production code smell**
- **“If (testing) then .....”**
- **Equality pollution**

Outline

- Bad Smells in Test Code
- Key Success Factors
- Terminology
- Key Test Refactorings
- Other Kinds of Test Smells

A Sobering Thought

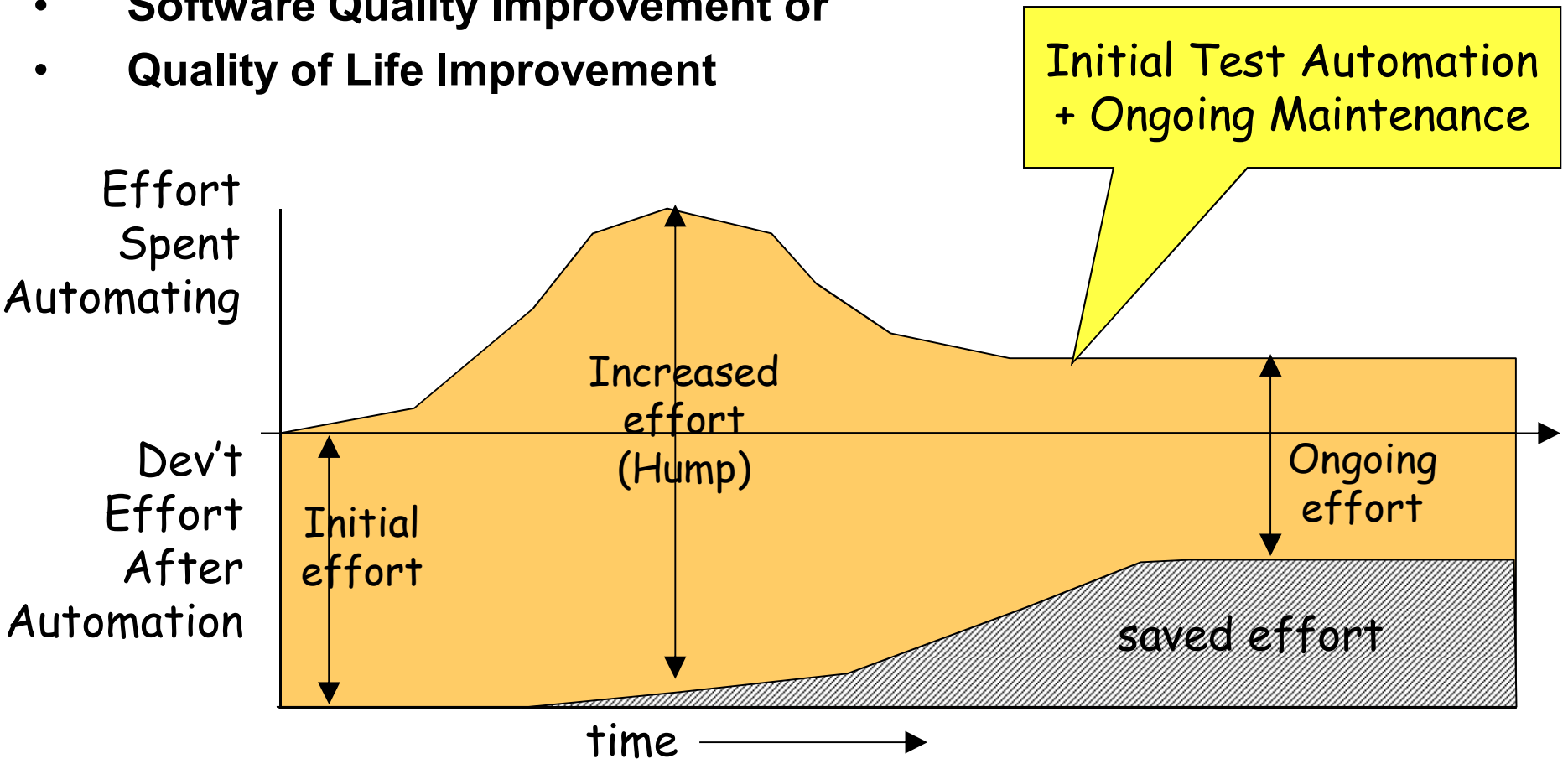
Expect to have just as much test code as production code!

The Challenge: How To Prevent Doubling Cost of Software Maintenance?

Economics of Maintainability

Test Automation is a lot easier to sell on

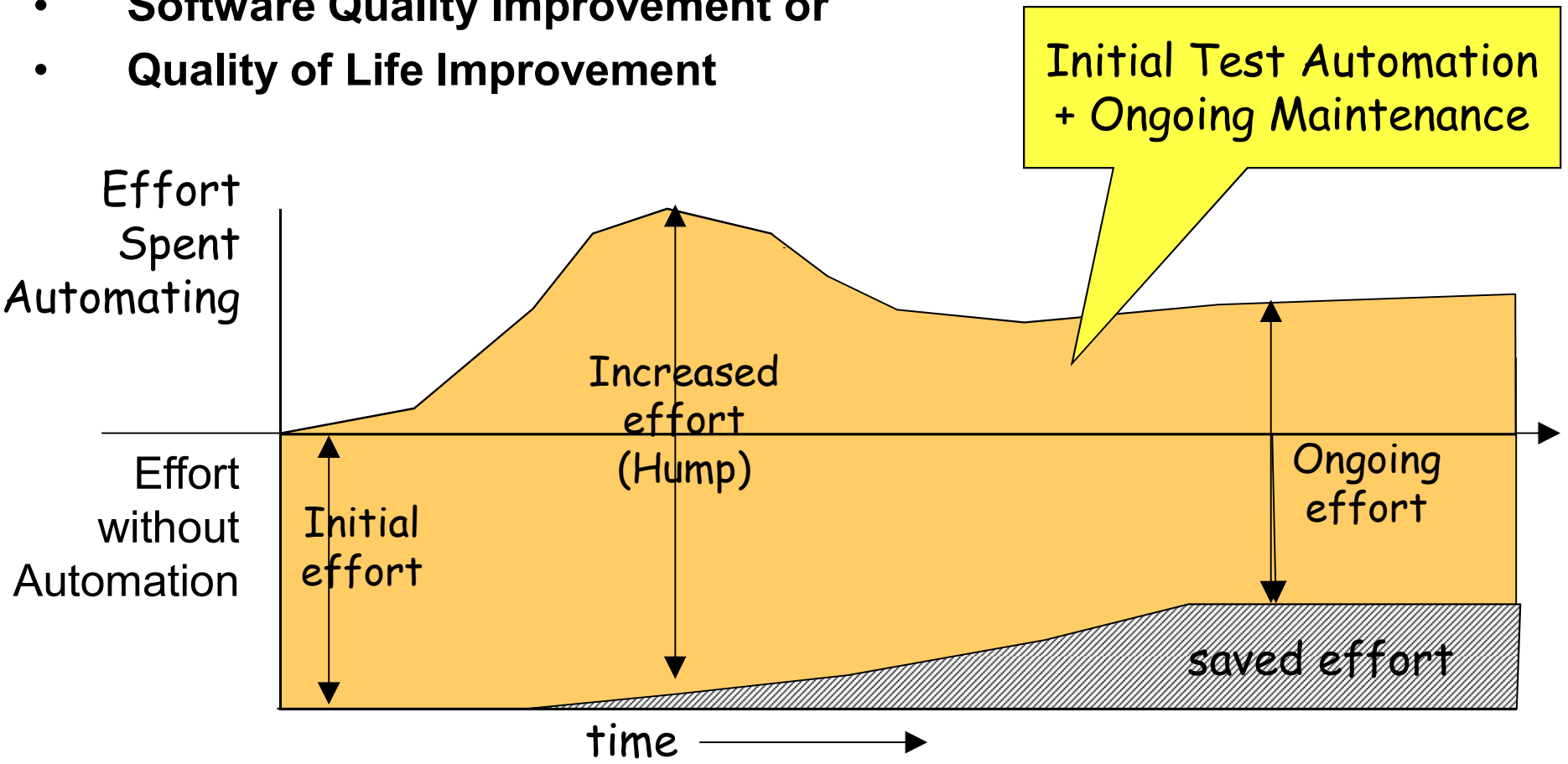
- **Cost reduction than**
- **Software Quality Improvement or**
- **Quality of Life Improvement**



Economics of Maintainability

Test Automation is a lot easier to sell on

- **Cost reduction than**
- **Software Quality Improvement or**
- **Quality of Life Improvement**

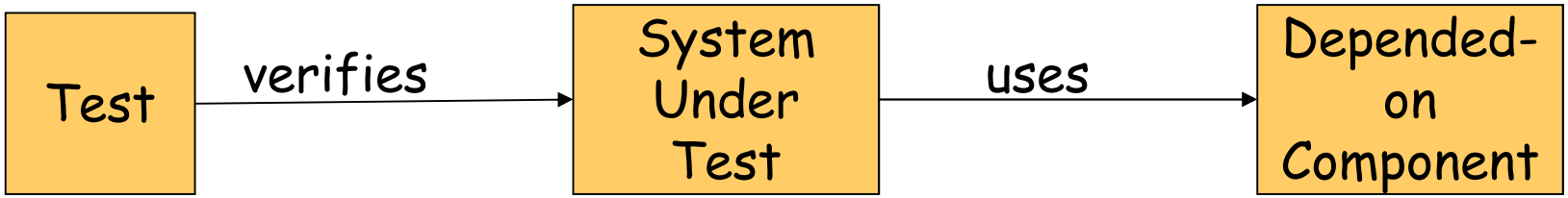


Outline

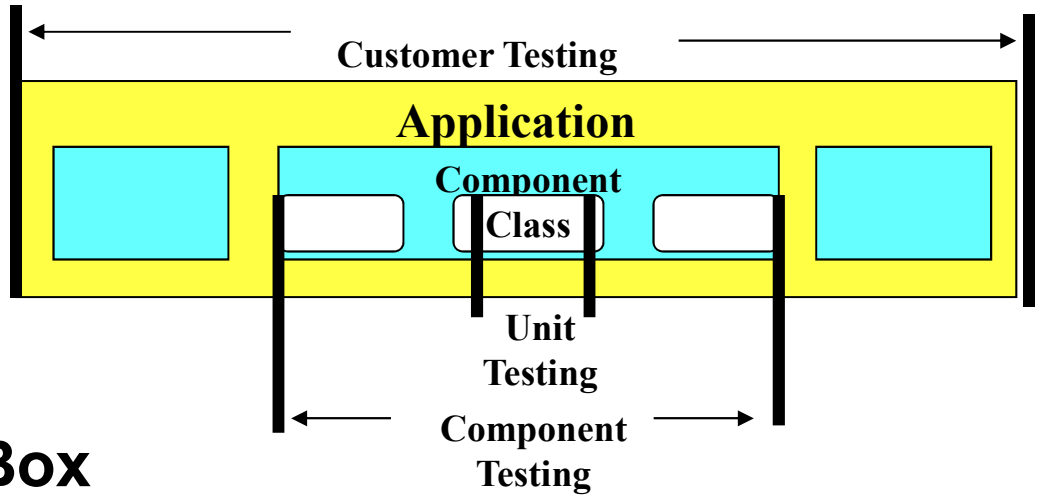
- Bad Smells in Test Code
- Key Success Factors
- Terminology
- Key Test Refactorings
- Other Kinds of Test Smells

Testing Terminology

- **Test vs SUT vs DOC:**



- **Unit vs Component vs Full System (Customer) Testing**



- **Black Box vs White Box**
  - **Black box: know <u>what</u> SUT should do**
  - **White box: know <u>how</u> SUT is built inside**

What is a Test Condition?

Given that the SUT is in state S

And depended-on component D is in state d (optional)

Pre-conditions (fixture, context)

When the client does X

– E.g. calls method X with arguments x1,x2, etc.

Exercise SUT, Action

Then

– SUT should respond to the client with Y
  » **Function return value**
  » **Updated argument**
  » **Exception thrown**
– And have the side effect of requesting Z from object O
  » **(optional)**
– And end up in state S'
  » **(stateful objects only)**

Post-conditions (expected results)

Also called Arrange, Act, Assert

Key Test Refactorings

- **Extract Method**
  - Use your IDE to convert several lines of code into one method call
- **Introduce Parameter**
  - When a the code extracted differs only in constants used
- **Convert Constant to Local**
  - To force Extract Method to pass it as a parameter so it can be varied from each calling location
- **Extract Method**
  - Let your IDE find other occurences of the same lines of code to call the same method

Key Test Refactorings (2)

- **Rename Method**
  - Align with intent. Repeat until all you have left is Given-When-Then
- **Inline Method**
  - Sometimes you have to inline code so that you can extract it in smaller chunks to match the domain
  - varied from each calling location
- **Extract Method**
  - Repeat until all you have left is Given-When-Then

(Re)Naming as a Process

- **Missing**
- **Nonsense**
- **Honest**
- **Honest and Complete**
- **Does the Right Thing**
- **Intent**
- **Domain Abstraction**

http://arlobelshee.com/good-naming-is-a-process-not-a-single-step/

Four-Phase Test

```
public void testAddItemQuantity_severalQuantity () {
    // Setup     or  // Arrange
    final int QUANTITY = 5 ;
    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice();
    // Exercise or // Act
    invoice.addItemQuantity(product, QUANTITY);
    // Verify    or // Assert
    LineItem expectedLineItem =    newLineItem(invoice,
        product, QUANTITY, product.getPrice()*QUANTITY );
    assertExactlyOneLineItem(invoice, expectedLineItem );
}  // Teardown
     // Shouldn't be needed
```

- Bill Wake
  http://xp123.com/articles/3a-arrange-act-assert/

This terminology reinforces our focus on mechanics, not intent!

Four-Phase Test

```
public void testAddItemQuantity_severalQuantity () {
    // Setup      or  // Arrange
    final int QUANTITY = 5 ;
    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice();
    // Exercise  or // Act
    invoice.addItemQuantity(product, QUANTITY);
    // Verify    or // Assert
    LineItem expectedLineItem =    newLineItem(invoice,
        product, QUANTITY, product.getPrice()*QUANTITY );
    assertExactlyOneLineItem(invoice, expectedLineItem );
}
```

Given an empty invoice

when I call addItemQuantity

Then the invoice will end up with exactly 1 lineItem on it.

- Use Domain-Specific Language
- Say Only What is Relevant

Improving Terminology

```
public void testAddItemQuantity_severalQuantity () {
    // Given
    final int QUANTITY = 5 ;
    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice();
    // When
    invoice.addItemQuantity(product, QUANTITY);
    // Then
    LineItem expectedLineItem =    newLineItem(invoice,product,
        QUANTITY, product.getPrice()*QUANTITY );
    assertExactlyOneLineItem(invoice, expectedLineItem );
}
```

- Use Domain-Specific Language
- Say Only What is Relevant

Improving Terminology

```
@Test public void
testAddItemQuantity_severalQuantity () {
    final int QUANTITY = 5 ;
    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice();
    // When
    invoice.addItemQuantity(product, QUANTITY);
    // Then
    LineItem expectedLineItem =    newLineItem(invoice, product,
        QUANTITY, product.getPrice()*QUANTITY );
    assertExactlyOneLineItem(invoice, expectedLineItem );
}
```

- Use Domain-Specific Language
- Say Only What is Relevant

Improving Terminology

```
@Test public void
addItem_severalQuantity_itemValueIsQuantityTimesProductPrice(){
    final int QUANTITY = 5 ;
    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice();
    // When
    invoice.addItemQuantity(product, QUANTITY);
    // Then
    LineItem expectedLineItem =    newLineItem(invoice, product,
        QUANTITY, product.getPrice()*QUANTITY );
    assertExactlyOneLineItem(invoice, expectedLineItem );
}
```

Constantly Strive to Improve Readability

- Use Domain-Specific Language
- Say Only What is Relevant

Improving Terminology

```
@Test public void
addItem_severalQuantity_itemValueIsQuantityTimesProductPrice(){
    final int QUANTITY = 5 ;
    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice();
    // When
    invoice.addItemQuantity(product, QUANTITY);
    // Then
    shouldBeExactlyOneLineItemOn(invoice,
        expectedLineItem(invoice, product, QUANTITY,
                         product.getPrice()*QUANTITY)   );
}
```

Constantly Strive to Improve Readability

- Use Domain-Specific Language
- Say Only What is Relevant

Improving Terminology

```
@Test public void
addItem_severalQuantity_itemValueIsQuantityTimesProductPrice(){
    final int QUANTITY = 5 ;
    Product product = createIrrelevantProduct();
    Invoice invoice = createIrrelevantInvoice();
    // When
    invoice.addItemQuantity(product, QUANTITY);
    // Then
    shouldBeExactlyOneLineItemOn(invoice,
        expectedLineItem(invoice, product, QUANTITY,
                         product.getPrice()*QUANTITY)   );
}
```

Constantly Strive to Improve Readability

- Use Domain-Specific Language
- Say Only What is Relevant

Improving Terminology

```
@Test public void
addItem_severalQuantity_itemValueIsQuantityTimesProductPrice(){
    final int QUANTITY = 5 ;
    Product product = givenAnyProduct();
    Invoice invoice = givenAnEmptyInvoice();
    // When
    invoice.addItemQuantity(product, QUANTITY);
    // Then
    shouldBeExactlyOneLineItemOn(invoice,
        expectedLineItem(invoice, product, QUANTITY,
                         product.getPrice()*QUANTITY)   );
}
```

Naming as a Process - Arlo Belshee

- Use Domain-Specific Language
- Say Only What is Relevant

Test Coverage

```
TestInvoiceLineItems {
    testAddItemQuantity_singleQuantity()
    testAddItemQuantity_severalQuantity{..}
    testAddItemQuantity_duplicateProduct {..}
    testAddItemQuantity_differentProduct () {..}
    testAddItemQuantity_zeroQuantity {..}
    testAddItemQuantity_severalQuantity_... {..}
    testAddItemQuantity_discountedPrice_... {..}
    testRemoveItem_noItemsLeft… {..}
    testRemoveItem_oneItemLeft… {..}
    testRemoveItem_ severalItemsLeft… {..}
}
```

Test Coverage

```
TestInvoiceLineItems {
    addItem_singleQuantity_itemValueIsProductPrice
    addItem_severalQuantity_itemValueIsQuantityTimesPr…
    addItem_duplicateProduct_singleItemHasSumOfQuantity
    addItem_differentProduct_oneItemPerProduct
    addItem_zeroQuantity_noItemAdded
    addItem_customerWithDiscount_itemValueIsDiscounted
    removeItem_onlyItem_noItemsLeft…
    removeItem_severalItems_oneLessItemLeft
    removeItem_severalItems_severalItemsLeft
}
```

Complete refactoring example at http://singapore2016.xunitpatterns.com/

Outline

- **Bad Smells in Test Code**
- **Key Success Factors**
- **Terminology**
- **Key Test Refactorings**
- **Other Kinds of Test Smells**

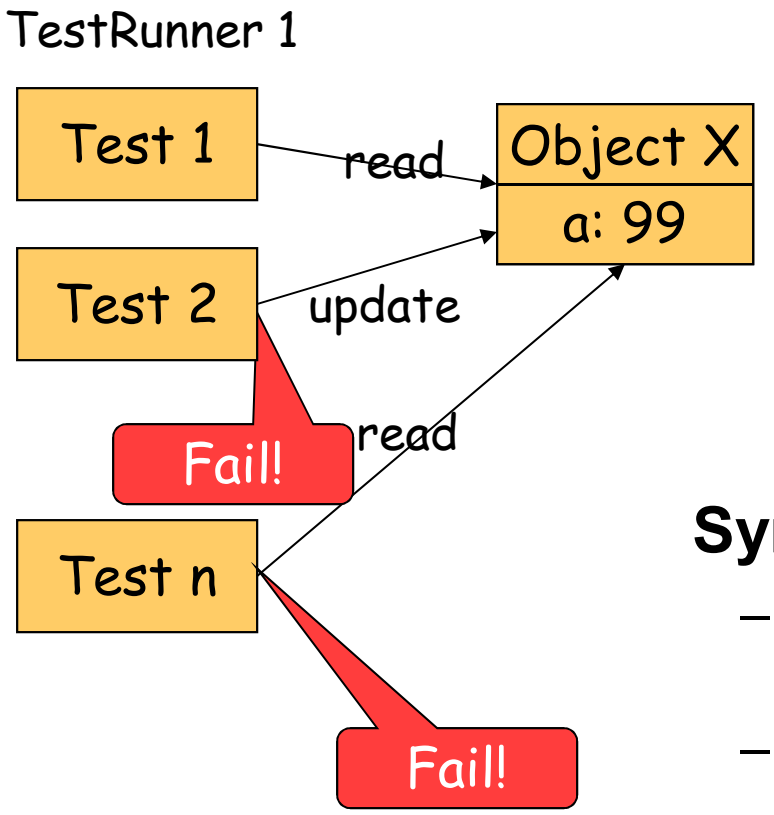
Behavior Smells (1)

- **Fragile Tests**
  - Every time you change the SUT, tests won't compile or start failing
  - You need to modify lots of tests to get things "Green" again
  - Greatly increases the cost of maintaining the system
- **Fragile Fixture**
  - Tests start failing when a shared fixture is modified
  - e.g. New records are put into the database

Behavior Smells (2)

- **Interacting Tests**
  - When one test fails, a bunch of other tests fail for no apparent reason because they depend on other tests' side effects
  - Tests cannot be run alone and are hard to maintain
- **Unrepeatable Tests**
  - Tests can't be run repeatedly without intervention
  - Caused by corruption of test fixture by other tests not cleaning up
- **Test Run War**
  - Seemingly random, transient test failures
  - Only occurs when several people testing simultaneously
  - Caused by parallel tests interacting with each other

Behavior Smell – Interacting Tests



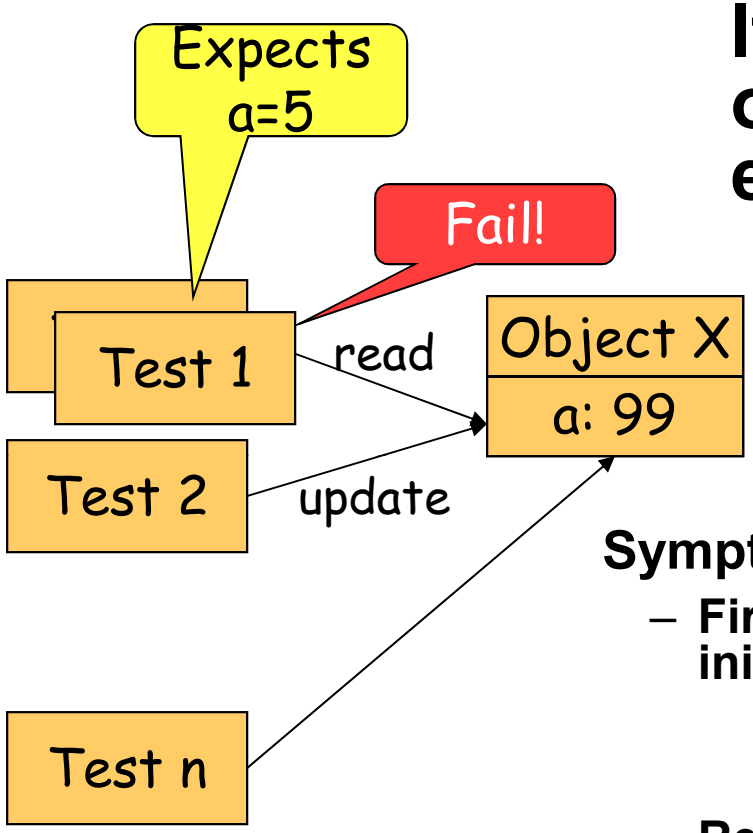
If many tests use same objects, tests can affect each other's results.

- Test 2 failure may leave Object X in state that causes Test n to fail.

Symptoms:

- **Tests that work by themselves failing when run in a suite.**
- **Cascading errors caused by a single bug failing a single test.**
  - » **Bug need not affect other tests directly but leaves fixture in wrong state for subsequent tests to succeed.**

Behavior Smell – Unrepeatable Tests



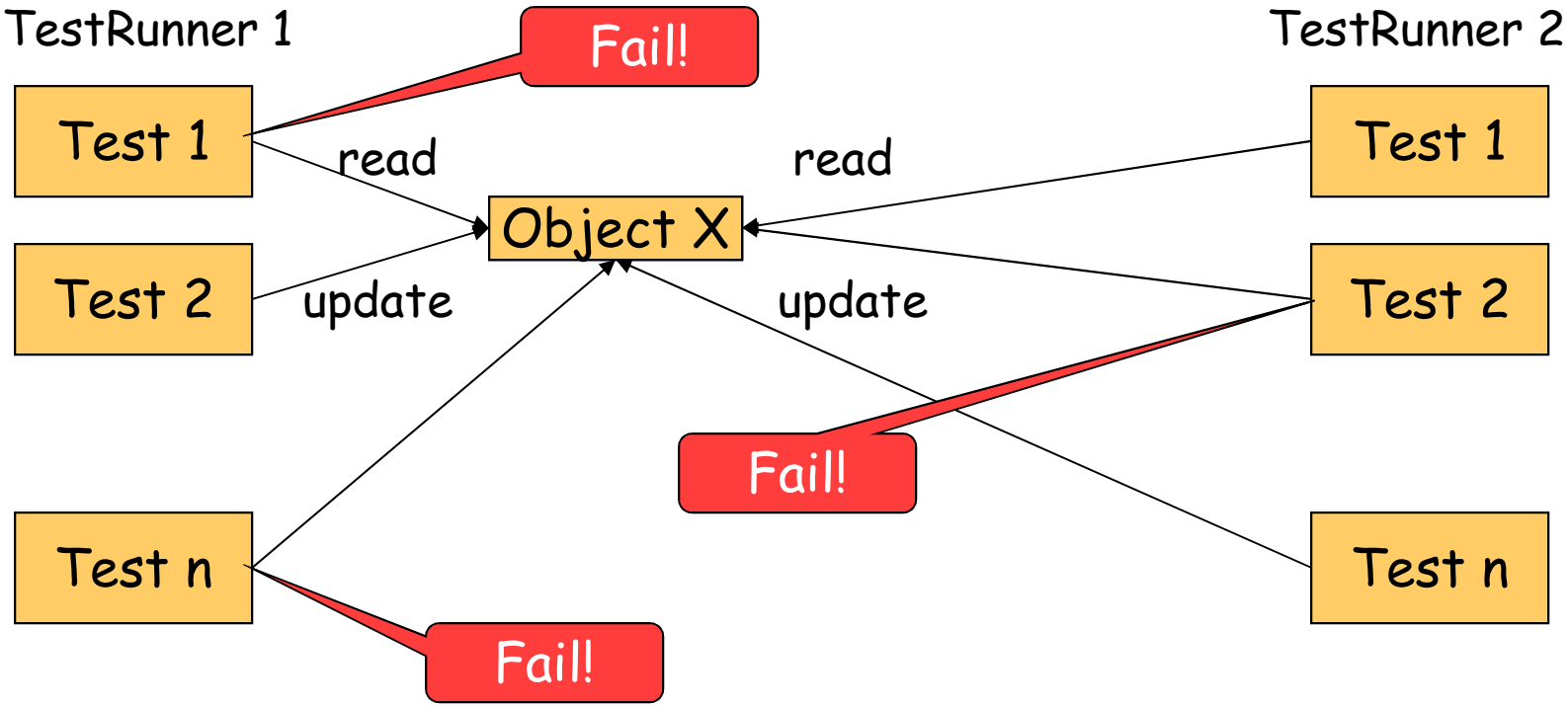
If many test runs use same objects, test runs can affect each other's results.

- Test 2 update may leave Object X in state that causes Test 1 to fail on next run.

Symptoms:

- **First run after opening the TestRunner or re-initing Shared Fixture behaves differently**
  - » **Succeed, Fail, Fail, Fail**
  - » **Fail, Succeed, Succeed, Succeed**
- **Resetting the fixture may "reset" things to square 1 (restarting the cycle)**
  - » **Closing and reopening the test runner for in-memory fixture**
  - » **Reinitializing the database**

Behavior Smell – Test Run War



- **If many test runners use the same objects (from Global Fixture), random results can occur.**
  - Interleaving of tests from parallel runners makes determining cause very difficult