xUnit Test Patterns and Smells

xUnit Test Patterns and Smells
Refactoring Test Code to Improve ROI

or:
Clean Code 99 – Test Code

Gerard Meszaros
jaoo2009@xunitpatterns.com
http://www.xunitpatterns.com

My Background

• Software developer
• Development manager
• Project Manager
• Software architect
• OOA/OOD Mentor
• XP/TDD Mentor
• Agile PM Mentor
• Test Automation Consultant
• Lean/Agile Coach/Consultant
Original Motivation

- Started writing automated unit tests in 1996
  - Smalltalk plus homebuilt unit test framework
- Started doing formal XP in 2000 right after 1st XP book came out
- Ran into cost of change problem with our tests
  - Up to 90% of effort was in changing tests
- Many teams I’ve visited have exhibited similar problems
  - Poorly written tests made automated unit testing scale poorly.

Material Background

- Started with 2 days onsite training
  - Java & JUnit
- Captured as draft of book
  - http://xunitPatterns.com
- Fine-tuned through
  - many years of practice
  - and delivery as half-day tutorial
- Published as book in 2007
- Continues to evolve
  - Understanding increases
  - Onsite training 2 -> 3 days, C#, C++
  - Tutorial ½ -> ¼ day
Objectives of this Talk

- Understand why maintainability of test code is important
- Be aware of Test Smells as symptoms of issues in your test code.
- Be aware of test design patterns that can address or prevent these issues

Agenda

- Introduction
  - Economics of Maintainability
  - Intro to Test Smells & Patterns
  - Refactoring Smelly Test Code
  - Writing New Tests Quickly
  - Wrap Up
Terminology

• Unit vs Component vs Customer Testing

  ![Diagram showing Application, Component, Class, Unit Testing, and Component Testing]

• Black Box vs White Box
  – Black box: know what it should do
  – White box: know how it is built inside

• Even Unit Tests should be black box.

xUnit Test Patterns and Smells

Terminology

• Test vs SUT vs DOC:

  ![Diagram showing Test, System Under Test, and Fixture or Context]

• Test vs. Spec. vs. Example
  – Doesn’t matter what you call them,
  – same smells and patterns apply!
**What Does it Take To be Successful?**

**Programming Experience**
+ xUnit Experience
+ Testing experience

---

Robust Automated Tests

---

**Why We Automate Unit Tests**

- **Self-Testing Code helps us:**
  - Produce better quality software
  - Produce the right software
  - Work faster
  - Respond to change (agility)

- **It does this by:**
  - Providing focus
  - Providing rapid feedback
  - Reducing stress levels (anxiety) by making changes safer

---

Requires writing tests *before* code (TDD)
A Sobering Thought

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

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

Agenda

- Introduction
  - Economics of Maintainability
    - Intro to Test Smells & Patterns
    - Refactoring Smelly Test Code
    - Writing New Tests Quickly
  - Wrap Up
Why is Test Maintainability so Crucial?

- Tests need to be maintained along with rest of the software.
- Testware must be much easier to maintain than production software, otherwise:
  - It will slow you down
  - It will get left behind
  - Value drops to zero
  - You’ll go back to manual testing

Critical Success Factor:
Writing tests in a maintainable style

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

Initial Test Automation + Ongoing Maintenance
Goals of Automated Developer Tests

• Before code is written
  – Tests as Specification
• After code is written
  – Tests as Documentation
  – Tests as Safety Net (Bug Repellent)
  – Defect Triangulation (Minimize Debugging)

• Minimize Cost of Running Tests
  – Fully Automated Tests
  – Repeatable Tests
  – Robust Tests

Agenda

• Introduction
• Economics of Maintainability
• Intro to Test Smells & Patterns
• Refactoring Smelly Test Code
• Writing New Tests Quickly
• Wrap Up
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”
Kinds of Test Smells

- Three common kinds of Test Smells:
  - 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

What’s a “Test Pattern”? 

- A “test pattern” is a recurring solution to a test automation problem
  - E.g. A “Mock Object” solves the problem of verifying the behavior of an object that should delegate behavior to other objects

- Invented in parallel by many people
Test Patterns occur at many levels:

- **Test Automation Strategy Patterns**
  - Recorded Test vs Scripted Test

- **Test Design Patterns**
  - Implicit SetUp vs Delegated SetUp

- **Test Coding Patterns**
  - Assertion Method, Creation Method

- **Language-specific Test Coding Idioms**
  - E.g., Expected Exception Test
    - Try/Catch
    - AssertThrows aBlock, expectedEx
    - ExpectedException attribute/annotation

---

**Agenda**

- Introduction
- Economics of Maintainability
- Intro to Test Smells & Patterns
- Refactoring Smelly Test Code
- Writing New Tests Quickly
- Wrap Up
What’s a Code Smell?
A problem visible when looking at test code:

- Tests are hard to understand
- Tests contain coding errors that may result in
  - Missed bugs
  - Erratic Tests
- Tests are difficult or impossible to write
  - No test API on SUT
  - Cannot control initial state of SUT
  - Cannot observe final state of SUT
- Sniff Test:
  - Problem must be visible (in their face) to test automater or test reader

Common Code Smells

- Conditional Test Logic
- Hard to Test Code
- Obscure Test
- Test Code Duplication
- Test Logic in Production
Example

- Test addItemQuantity and removeLineItem methods of Invoice

```java
public void testAddItemQuantity_severalQuantity() throws Exception {
    try {
        // Setup Fixture
        final int QUANTITY = 5;
        Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
        Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
        Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"), billingAddress, shippingAddress);
        Invoice invoice = new Invoice(customer);
        // Exercise SUT
        invoice.addItemQuantity(product, QUANTITY);
        // Verify Outcome
        List lineItems = invoice.getLineItems();
        if (lineItems.size() == 1) {
            LineItem actualLineItem = (LineItem) lineItems.get(0);
            assertEquals(invoice, actualLineItem.getInvoice());
            assertEquals(product, actualLineItem.getProduct());
            assertEquals(quantity, actualLineItem.getQuantity());
            assertEquals(new BigDecimal("30"), actualLineItem.getPercentDiscount());
            assertEquals(new BigDecimal("19.99"), actualLineItem.getUnitPrice());
            assertEquals(new BigDecimal("69.96"), actualLineItem.getExtendedPrice());
        } else {
            assertTrue("Invoice should have exactly one line item", false);
        }
    } finally {
        deleteObject(expectedLineItem);
        deleteObject(invoice);
        deleteObject(product);
        deleteObject(customer);
        deleteObject(billingAddress);
        deleteObject(shippingAddress);
    }
}
```
Verifying the Outcome

List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertEquals(invoice, actualLineItem.getInvoice());
    assertEquals(product, actualLineItem.getProduct());
    assertEquals(quantity, actualLineItem.getQuantity());
    assertEquals(new BigDecimal("30"), actualLineItem.getPercentDiscount());
    assertEquals(new BigDecimal("19.99"), actualLineItem.getUnitPrice());
    assertEquals(new BigDecimal("69.96"), actualLineItem.getExtendedPrice());
} else {
    assertTrue("Invoice should have exactly one line item", false);
}

Use Better Assertion

List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertEquals(invoice, actualLineItem.getInvoice());
    assertEquals(product, actualLineItem.getProduct());
    assertEquals(quantity, actualLineItem.getQuantity());
    assertEquals(new BigDecimal("30"), actualLineItem.getPercentDiscount());
    assertEquals(new BigDecimal("19.99"), actualLineItem.getUnitPrice());
    assertEquals(new BigDecimal("69.96"), actualLineItem.getExtendedPrice());
} else {
    fail("invoice should have exactly one line item");
}
Use Better Assertion

List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    assertEquals(invoice, actualLineItem.getInvoice());
    assertEquals(product, actualLineItem.getProduct());
    assertEquals(quantity, actualLineItem.getQuantity());
    assertEquals(new BigDecimal("30"),
        actualLineItem.getPercentDiscount());
    assertEquals(new BigDecimal("19.99"),
        actualLineItem.getUnitPrice());
    assertEquals(new BigDecimal("69.96"),
        actualLineItem.getExtendedPrice());
} else {
    fail("invoice should have exactly one line item");
}

Expected Object

List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    LineItem expectedLineItem =
        newLineItem(invoice, product, QUANTITY);
    assertEquals(expectedLineItem.getInvoice(),
        actualLineItem.getInvoice());
    assertEquals(expectedLineItem.getProduct(),
        actualLineItem.getProduct());
    assertEquals(expectedLineItem.getQuantity(),
        actualLineItem.getQuantity());
    assertEquals(expectedLineItem.getPercentDiscount(),
        actualLineItem.getPercentDiscount());
    assertEquals(expectedLineItem.getUnitPrice(),
        actualLineItem.getUnitPrice());
    assertEquals(expectedLineItem.getExtendedPrice(),
        actualLineItem.getExtendedPrice());
} else {
    fail("invoice should have exactly one line item");
}
**Expected Object**

```java
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY, product.getPrice()*QUANTITY);
    assertEquals(expectedLineItem.getInvoice(), actualLineItem.getInvoice());
    assertEquals(expectedLineItem.getProduct(), actualLineItem.getProduct());
    assertEquals(expectedLineItem.getQuantity(), actualLineItem.getQuantity());
    assertEquals(expectedLineItem.getPercentDiscount(), actualLineItem.getPercentDiscount());
    assertEquals(expectedLineItem.getUnitPrice(), actualLineItem.getUnitPrice());
    assertEquals(expectedLineItem.getExtendedPrice(), actualLineItem.getExtendedPrice());
} else {
    fail("invoice should have exactly one line item");
}
```

**Introduce Custom Assert**

```java
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY, product.getPrice()*QUANTITY);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
} else {
    fail("invoice should have exactly one line item");
}
```
### Introduce Custom Assert

```java
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice,
                                            product, QUANTITY, product.getPrice() * QUANTITY);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
} else {
    fail("invoice should have exactly one line item");
}
```

### Replace Conditional Logic with Guard Assertion

```java
List lineItems = invoice.getLineItems();
assertEquals("number of items", lineItems.size(), 1);
LineItem actualLineItem = (LineItem) lineItems.get(0);
LineItem expectedLineItem = newLineItem(invoice,
                                         product, QUANTITY, product.getPrice() * QUANTITY);
assertLineItemsEqual(expectedLineItem, actualLineItem);
```
The Whole Test

```java
public void testAddItemQuantity_severalQuantity() throws Exception {
    try {
        // Setup Fixture
        final int QUANTITY = 5;
        Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
        Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
        Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"), billingAddress, shippingAddress);
        Invoice invoice = new Invoice(customer);
        // Exercise SUT
        invoice.addItemQuantity(product, QUANTITY);
        // Verify Outcome
        List lineItems = invoice.getLineItems();
        assertEquals("number of items", lineItems.size(), 1);
        LineItem actualLineItem = (LineItem) lineItems.get(0);
        LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
        assertLineItemsEqual(expectedLineItem, actualLineItem);
    } finally {
        deleteObject(expectedLineItem);
        deleteObject(invoice);
        deleteObject(product);
        deleteObject(customer);
        deleteObject(billingAddress);
    }
}
```
**xUnit Test Patterns and Smells**

### Inline Fixture Teardown – Naive
```java
public void testAddItemQuantity_severalQuantity() { 
    try { 
        // Setup Fixture 
        // Exercise SUT 
        // Verify Outcome 
    } finally { 
        deleteObject(expectedLineItem);
        deleteObject(invoice);
        deleteObject(product);
        deleteObject(customer);
        deleteObject(billingAddress);
        deleteObject(shippingAddress);
    }
}
```

### Inline Fixture Teardown – Robust
```java
public void testAddItemQuantity_severalQuantity() { 
    try { 
        // Setup Fixture 
        // Exercise SUT 
        // Verify Outcome 
    } finally { 
        try { 
            deleteObject(expectedLineItem);
        } finally { 
            try { 
                deleteObject(invoice);
            } finally { 
                try { 
                    deleteObject(product);
                } finally { 
                    deleteObject(customer);
                    deleteObject(billingAddress);
                    deleteObject(shippingAddress);
                }
            }
        }
    }
```
**Implicit Fixture Teardown – Naive**

```java
public void testAddItemQuantity_severalQuantity() {
    // Setup Fixture
    // Exercise SUT
    // Verify Outcome
}

public void tearDown() {
    deleteObject(expectedLineItem);
    deleteObject(invoice);
    deleteObject(product);
    deleteObject(customer);
    deleteObject(billingAddress);
    deleteObject(shippingAddress);
}
```

**Implicit Fixture Teardown – Robust**

```java
public void testAddItemQuantity_severalQuantity() {
    // Setup Fixture
    // Exercise SUT
    // Verify Outcome
}

public void tearDown() {
    try {
        deleteObject(expectedLineItem);
    } finally {
        try {
            deleteObject(invoice);
        } finally {
            try {
                deleteObject(product);
            } finally {
                // ...
            }
        }
    }
}
Automated Fixture Teardown

```java
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;
    Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    addTestObject(billingAddress);
    Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    addTestObject(shippingAddress);
}
```

```java
public void tearDown() {
    deleteAllTestObjects();
}
```

Automated Fixture Teardown

```java
public void deleteAllTestObjects() {
    Iterator i = testObjects.iterator();
    while (i.hasNext()) {
        try {
            Deletable object = (Deletable) i.next();
            object.delete();
        } catch (Exception e) {
            // do nothing if the remove failed
        }
    }
}
```
**xUnit Test Patterns and Smells**

**Transaction Rollback Teardown**

```java
public void setUp() {
    TransactionManager.beginTransaction();
}

public void tearDown() {
    TransactionManager.abortTransaction();
}
```

Important: SUT must not commit transaction
– DFT Pattern: Humble Transaction Controller

---

**The Whole Test**

```java
public void testAddItemQuantity_severalQuantity() throws Exception {
    // Setup Fixture
    final int QUANTITY = 5;
    Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    addTestObject(billingAddress);
    Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    addTestObject(shippingAddress);
    Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"), billingAddress, shippingAddress);
    addTestObject(shippingAddress);
    addTestObject(shippingAddress);
    Invoice invoice = new Invoice(customer);
    addTestObject(shippingAddress);
    // Exercise SUT
    invoice.addItemQuantity(product, QUANTITY);
    // Verify Outcome
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    LineItem expectedLineItem = new LineItem(invoice, product, QUANTITY);
    assertEquals(expectedLineItem, actualLineItem);
}
```

// No Visible Fixture Tear Down!
The Whole Test

```java
public void testAddItemQuantity_severalQuantity() throws Exception {
    // Setup Fixture
    final int QUANTITY = 5;
    Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    addTestObject(billingAddress);
    Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    addTestObject(billingAddress);
    Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"), billingAddress, shippingAddress);
    addTestObject(billingAddress);
    addTestObject(billingAddress);
    Invoice invoice = new Invoice(customer);
    addTestObject(billingAddress);
    
    // Exercise SUT
    invoice.addItemQuantity(product, QUANTITY);
    // Verify Outcome
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```

Hard-Coded Test Data

```java
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;
    Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"), billingAddress, shippingAddress);
    Invoice invoice = new Invoice(customer);
    // Exercise SUT
    invoice.addItemQuantity(product, QUANTITY);
}
Distinct Generated Values

```java
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;
    Address billingAddress = new Address(getUniqueString(),
                                         getUniqueString(),
                                         getUniqueString(),
                                         getUniqueString(),
                                         getUniqueString());
    Address shippingAddress = new Address(getUniqueString(),
                                         getUniqueString(),
                                         getUniqueString(),
                                         getUniqueString(),
                                         getUniqueString());
    Customer customer = new Customer(
        getUniqueInt(),
        getUniqueString(),
        getUniqueString(),
        getUniqueDiscount(),
        billingAddress, shippingAddress);
    Product product = new Product(
        getUniqueInt(),
        getUniqueString(),
        getUniqueNumber());
    Invoice invoice = new Invoice(customer);
}
```
**xUnit Test Patterns and Smells**

### Creation Method

```java
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;
    Address billingAddress = createAnonymousAddress();
    Address shippingAddress = createAnonymousAddress();
    Customer customer = createCustomer(billingAddress, shippingAddress);
    Product product = createAnonymousProduct();
    Invoice invoice = new Invoice(customer);
    // Exercise
    invoice.addItemQuantity(product, QUANTITY);
    // Verify
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY, product.getPrice() * QUANTITY);
    List lineItems = invoice.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```

---

**xUnit Test Patterns and Smells**

### Obscure Test - Irrelevant Information

```java
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;
    Address billingAddress = createAnonymousAddress();
    Address shippingAddress = createAnonymousAddress();
    Customer customer = createCustomer(billingAddress, shippingAddress);
    Product product = createAnonymousProduct();
    Invoice invoice = new Invoice(customer);
    // Exercise
    invoice.addItemQuantity(product, QUANTITY);
    // Verify
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY, product.getPrice() * QUANTITY);
    List lineItems = invoice.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;

    Customer customer = createAnonymousCustomer();
    Product product = createAnonymousProduct();
    Invoice invoice = new Invoice(customer);
    // Exercise
    invoice.addItemQuantity(product, QUANTITY);
    // Verify
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY, product.getPrice() * QUANTITY);
    List lineItems = invoice.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}

Introduce Custom Assertion

public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;

    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice()
        // Exercise
    invoice.addItemQuantity(product, QUANTITY);
        // Verify
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY, product.getPrice() * QUANTITY);
    List lineItems = invoice.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}

Refactoring

Introduce Custom Assertion

public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;

    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice()
        // Exercise
    invoice.addItemQuantity(product, QUANTITY);
        // Verify
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY, product.getPrice() * QUANTITY);
    List lineItems = invoice.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertExactlyOneLineItem(invoice, expectedLineItem);
}

Introduce Custom Assertion

public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;

    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice()
        // Exercise
    invoice.addItemQuantity(product, QUANTITY);
        // Verify
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY, product.getPrice() * QUANTITY);
    List lineItems = invoice.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
    assertExactlyOneLineItem(invoice, expectedLineItem);
}
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;
    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice();
    // Exercise
    invoice.addItemQuantity(product, QUANTITY);
    // Verify
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY, product.getPrice() * QUANTITY);
    assertExactlyOneLineItem(invoice, expectedLineItem);
}

This Looks a Lot Like Keyword-Driven Testing

Same Underlying Principles:
- Use Domain-Specific Language
- Say Only What is Relevant

Agenda
- Introduction
- Economics of Maintainability
- Intro to Test Smells & Patterns
- Refactoring Smelly Test Code
- Writing New Tests Quickly
- Behaviour Smells
- Wrap Up
Test Coverage

TestInvoiceLineItems extends TestCase {
    TestAddItemQuantity_oneItem { .. }
    TestAddItemQuantity_severalItems { .. }
    TestAddItemQuantity_duplicateProduct { .. }
    TestAddItemQuantity_zeroQuantity { .. }
    TestAddItemQuantity_severalQuantity { .. }
    TestAddItemQuantity_discountedPrice { .. }
    TestRemoveItem_noItemsLeft { .. }
    TestRemoveItem_oneItemLeft { .. }
    TestRemoveItem_severalItemsLeft { .. }
}

Rapid Test Writing

public void testAddItemQuantity_duplicateProduct() {
    final int QUANTITY = 1;
    final int QUANTITY2 = 2;
    Product product1 = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice();
    // Exercise
    invoice.addItemQuantity(product1, QUANTITY);
    invoice.addItemQuantity(product1, QUANTITY2);
    // Verify
    LineItem expectedLineItem1 = newLineItem(invoice, product, QUANTITY + QUANTITY2, product.getPrice() * (QUANTITY + QUANTITY2));
    assertExactlyOneLineItem(invoice, expectedLineItem1);
}
xUnit Test Patterns and Smells

Test Coverage

TestInvoiceLineItems extends TestCase {
  TestAddItemQuantity_oneItem {..}
  TestAddItemQuantity_severalItems {..}
  TestAddItemQuantity_duplicateProduct {..}
  TestAddItemQuantity_zeroQuantity {..}
  TestAddItemQuantity_severalQuantity {..}
  TestAddItemQuantity_discountedPrice {..}
  TestRemoveItem_noItemsLeft {..}
  TestRemoveItem_oneItemLeft {..}
  TestRemoveItem_severalItemsLeft {..}
}

Rapid Test Writing

public void testAddItemQuantity_severalItems() {
  final int QUANTITY = 1;
  Product product1 = createAnonymousProduct();
  Product product2 = createAnonymousProduct();
  Invoice invoice = createAnonymousInvoice();
  // Exercise
  invoice.addItemQuantity(product1, QUANTITY);
  invoice.addItemQuantity(product2, QUANTITY);
  // Verify
  LineItem expectedLineItem1 = newLineItem(invoice,
      product, QUANTITY, product.getPrice() * QUANTITY);
  LineItem expectedLineItem2 = newLineItem(invoice,
      product2, QUANTITY, product2.getPrice() * QUANTITY);
  assertExactlyTwoLineItems(invoice,
      expectedLineItem1, expectedLineItem2);
}

"Pull" also applied to new Custom Assertion
Testability Patterns

• **Humble Object**
  – Objects closely coupled to the environment should not do very much (be humble)
  – Should delegate real work to a context-independent testable object

• **Dependency Injection**
  – Client "injects" depended-on objects into SUT
  – Tests can pass a Test Double to control "indirect inputs" from dependents

• **Dependency Lookup**
  – SUT asks another object for it’s dependencies
  – Service Locator, Object Factory, Component Registry

• **Test-Specific Subclass**
  – Can extend the SUT to all access by test

---

Agenda

• **Introduction**
• **Economics of Maintainability**
• **Intro to Test Smells & Patterns**
• **Refactoring Smelly Test Code**
• **Writing New Tests Quickly**
  • **Behaviour Smells**
• **Wrap Up**
What’s a Behavior Smell

• A problem seen when running tests.
• Tests fail when they should pass
  – or pass when they should fail (rarer)
• The problem is with how tests are coded;
  – not a problem in the SUT
• Sniff Test:
  – Detectable via compile or execution behavior of tests

Common Behavior Smells

• Slow Tests
• Erratic Tests
  – Too many variants to list here
• Fragile Tests
  – The 4 sensitivities
• Assertion Roulette
• Frequent Debugging
• Manual Intervention
Slow Tests

- **Slow Tests**
  - It takes several minutes to hours to run all the tests

- **Impact**
  - Lost productivity caused by waiting for tests
  - Lost quality due to running tests less frequently

- **Causes:**
  - Slow Component Usage
    - e.g. Database
  - Asynchronous Test
    - e.g. Delays or Waits
  - General Fixture
    - too much fixture being setup

---

Shared Test Fixture

- **What it is:**
  - Improves test run times by reducing setup overhead.
  - A “standard” test environment applicable to all tests is built and the tests reuse the same fixture instance.
### Shared Test Fixture

**Variations:**
- Fixture is shared between some/all the tests in a single test run
- Fixture may be shared across many TestRunners (Global Text Fixture)

**Examples:**
- Standard Database contents
- Standard Set of Directories and Files
- Standard set of objects

---

### Erratic Tests

**Interacting Tests**
- When one test fails, a bunch of other tests fail for no apparent reason because they depend on other tests’ side effects

**Unrepeatable Tests**
- Tests can’t be run repeatedly without intervention

**Test Run War**
- Seemingly random, transient test failures
- Only occurs when several people testing simultaneously

**Resource Optimism**
- Tests depend on something in the environment that isn’t available

**Non-Deterministic Tests**
- Tests depend on non-deterministic inputs
**Erratic Tests – 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 fail 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.

---

**Erratic Tests – 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-initializing 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
 Erratic Tests – 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

Avoiding Erratic Tests - Fresh Fixture

• What it is:
  – “Brand new” fixture built for each test
  – Tests are completely independent
Agenda

• Introduction
• Economics of Maintainability
• Intro to Test Smells & Patterns
• Refactoring Smelly Test Code
• Writing New Tests Quickly
• Behaviour Smells

• Wrap Up

A Recipe for Success

1. Write some tests
   • start with the easy ones!
2. Note the Test Smells that show up
3. Refactor to remove obvious Test Smells
   • Apply appropriate xUnit Test Patterns
4. Write some more tests
   • possibly more complex
5. Repeat from Step 2 until:
   • All necessary tests written
   • No smells remain
What Next?

- You have a better idea of:
  - what can be achieved
  - problems to look for
    - Test Smells
      - symptoms (smells) vs root causes
- You have an initial list of patterns to address root causes
  - More at the web site and in the book
- Time to go “Smell Hunting”

Be Pragmatic!

- Not all Smells can (or should) be eliminated
  - Cost of having smell vs. cost of removing it
  - Cost to remove it now vs. cost of removing it later
- Catalog of Smells and Causes gives us the tools to make the decision intelligently
  - Trouble-shooting flow chart
  - Suggested Patterns for removing cause
- Catalog of Patterns gives us the tools to eliminate the Smells when we choose to do so
  - How it Works
  - When to Use It
  - Before/After Code samples
  - Refactoring notes
What **Does** it Take To be Successful?

Programing Experience
+ xUnit Experience
+ Testing Experience
+ Design for Testability
- Test Smells
+ Test Automation Patterns
+ Fanatical Attention to Test Maintainability
-------------------------------
= Robust, Maintainable Automated Tests

---

**xUnit Test Patterns and Smells**

More on xUnit Patterns & Smells

- Tutorial:  
  – Friday Morning
- Book:  
  xUnit Test Patterns
  Refactoring Test Code  
  – published by Addison Wesley
- Website:  
  http://xunitpatterns.com
- Onsite Training  
  – training at xunitpatterns.com  
  – +1-403-827-2967  
  – jaa02009@xunitpatterns.com

Thank You!  
Gerard
Questions & Comments?

Resources for Testing

Reminder:

Tutorial exercises and solutions available at:

http://tutorialexercises.xunitpatterns.com
http://tutorialssolutions.xunitpatterns.com
### Books on xUnit Test Automation

- **xUnit Test Patterns – Refactoring Test Code**  
  – Gerard Meszaros
- **Test-driven Development - A Practical Guide**  
  – David Astels
- **Test-driven Development - By Example**  
  – Kent Beck
- **Test-Driven Development in Microsoft .NET**  
  – James Newkirk, Alexei Vorontsov
- **Unit Testing With Java - How tests drive the code**  
  – Johannes Link
- **JUnit Recipes**  
  – J.B. Rainsberger

### Other Useful Books

- **Working Effectively with Legacy Code**  
  – Michael Feathers
- **Fit for Software Development**  
  – Rick Mugridge, Ward Cunningham
- **Refactoring - Improving the Design of Existing Code**  
  – Martin Fowler plus contributors
- **Design Patterns: Reusable Elements of Design**  
  – Erich Gamma, Richard Helm, Ralph Johnson, John Vlisides
### Coding Objectives Comparison

<table>
<thead>
<tr>
<th></th>
<th>Production</th>
<th>Testware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>Important</td>
<td>Crucial</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Important</td>
<td>Crucial</td>
</tr>
<tr>
<td>Execution Speed</td>
<td>Crucial</td>
<td>Somewhat</td>
</tr>
<tr>
<td>Reusability</td>
<td>Important</td>
<td>Somewhat</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Important</td>
<td>Not</td>
</tr>
<tr>
<td>Simplicity</td>
<td>Important?</td>
<td>Crucial</td>
</tr>
<tr>
<td>Ease of writing</td>
<td>Important?</td>
<td>Crucial</td>
</tr>
</tbody>
</table>