Instructor Biography

Gerard Meszaros is independent consultant specializing in agile development processes. Gerard built his first unit testing framework in 1996 and has been doing automated unit testing ever since. He is an expert in agile methods, test automation patterns, refactoring of software and tests, and design for testability.

Gerard has applied automated unit and acceptance testing on projects ranging from full-on eXtreme Programming to traditional waterfall development and technologies ranging from Java, Smalltalk and Ruby to PLSQL stored procedures and SAP’s ABAP. His first book xUnit Test Patterns - Refactoring Test Code was published in 2007. His current book (co-authored) is available in draft form at http://testingguidance.codeplex.com/

Gerard Meszaros
agile2009@xunitpatterns.com
Tutorial 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

Agenda

- Introduction
- Motivation
- Intro to Smells & Patterns
- Code Smells & Remedies
- Behavior Smells & Remedies
- Project Smells & Remedies *
- Wrap Up

* Denotes topics & slides skipped in 90 minute version
Objectives of Tutorial

• Understand why Test Smells are important
• Be able to recognize key code smells
• Be aware of test design patterns that can address or prevent these code smells
• Be able to recognize Behavior Smells and be aware of patterns to address them
• Be able to recognize Project Smells how they are related to Code and Behavior Smells

Half-day tutorial exercises and solutions available at:

• http://tutorialexercises.xunitpatterns.com
• http://tutorialsolutions.xunitpatterns.com

Outline

• Introduction

  • Motivation
    – Why is Test maintainability important?
    – How do we make tests maintainable?

• Intro to Smells & Patterns
• Code Smells & Remedies
• Behavior Smells & Remedies
• Project Smells & Remedies
• Wrap Up
**Terminology**

- **Test vs SUT vs DOC:**
  - Test: verifies
  - System Under Test: uses
  - Depended-on Component

- **Unit vs Component vs Customer Testing**

- **Black Box vs White Box**
  - Black box: know what it should do
  - White box: know how it is built inside

---

**What Does it Take To be Successful?**

Programming Experience

+ xUnit Experience

+ Testing experience

-------------------------------

Robust Automated Tests
**Why We Write 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)

---

**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>
A Sobering Thought

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

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

Why are They so Crucial?

• Tests need to be maintained along with rest of the software.

• Testware must be much easier to maintain than the 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

---

**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 Localization (Minimize Debugging)

- **Minimize Cost of Running Tests**
  - Fully Automated Tests
  - Repeatable Tests
  - Robust Tests
Outline

• Introduction
• Motivation

• Intro to Smells & Patterns
  – What is a Test Smell?
  – What is a Test Pattern?

• Code Smells & Remedies
• Behavior Smells & Remedies
• Project Smells & Remedies
• 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”
What's a “Test Smell”?  

- 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

- 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
    - Expected Exception Test, Constructor Test
xUnit Test Patterns and Smells

Outline

• Introduction
• Motivation
• Intro to Smells & Patterns
• Code Smells & Remedies
  – Refactoring a Smelly Test
  – Review of Test Patterns Used
• Behavior Smells & Remedies
• Project Smells & Remedies *
• 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
The Whole Test

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 {
    assertEquals("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");
}
**xUnit Test Patterns and Smells**

**Pattern**

**Expected Object**

```java
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");
}
```

**Verbose Test**

```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

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);
        deleteObject(shippingAddress);
    }
```

xUnit Test Patterns and Smells

Refactoring
**Pattern**

**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);
    }
}
```

---

**Pattern**

**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);
                }
            }
        }
    }
```
**xUnit Test Patterns and Smells**

**Pattern**

**Implicit Fixture Teardown - Naive**

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

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

---

**Pattern**

**Implicit Fixture Teardown - Robust**

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

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

**Pattern**

```java
public void testAddNewItemQuantity_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);
}

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

**Pattern**

```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

---

xUnit Test Patterns and Smells

**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(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 = 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**

**Pattern**

**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 customer1 = new Customer(
                 getUniqueInt(), getUniqueString(),
                 getUniqueString(), getUniqueDiscount(),
                 billingAddress, shippingAddress);
    Product product = new Product(
                   getUniqueInt(), getUniqueString(),
                   getUniqueNumber());
    Invoice invoice = new Invoice(customer);
}
```

**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);
}
```
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;

    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);
}

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);
}
xUnit Test Patterns and Smells

Introduce Custom Assertion

```java
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);
}
```

Refactoring

The Whole Test – Done

```java
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
**Test Coverage**

```java
TestInvoiceLineItems extends TestCase {
    TestAddItemQuantity_oneItem {..}
    TestAddItemQuantity_severalItems {..}
    TestAddItemQuantity_duplicateProduct {..}
    TestAddItemQuantity_zeroQuantity {..}
    TestAddItemQuantity_severalQuantity {..}
    TestAddItemQuantity_discountedPrice {..}
    TestRemoveItem_noItemsLeft {..}
    TestRemoveItem_oneItemLeft {..}
    TestRemoveItem_severalItemLeft {..}
}
```

**Rapid Test Writing**

```java
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,
        product, QUANTITY, product.getPrice() * QUANTITY);
    assertExactlyTwoLineItems(invoice,
        expectedLineItem1, expectedLineItem2);
}
```
Hard to Test Code

- Code can be hard to test for a number of reasons:
  - Too closely coupled to other software
  - No interface provided to set state, observe state
  - Only asynchronous interfaces provided

- Root Cause is lack of Design for Testability
  - Comes naturally with Test-Driven Development
  - Must be retrofitted to legacy (test-less) software

- Temporary Workaround is Test Hook
  - Becomes Test Logic in Production (code smell) if not removed

Test Double Patterns

- Replace depended-on components with test-specific ones to isolate SUT

- Kinds of Test Doubles
  - Test Stubs return test-specific values
  - Test Spies record method calls and arguments for verification by Test Method
  - Mock Objects verify the method calls and arguments themselves
  - Fake Objects provide (apparently) same services in a “lighter” way

- Test Doubles need to be “installed”
  - Dependency Injection
  - Dependency Lookup

- Configurable Test Doubles are reusable but need to be configure with test-specific values
  - return values
  - expected method calls & arguments
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

---

Test Logic in Production

**Test Hook:**

```
If (testing) then
  // test-specific logic
else
  // production logic
endif
```

**Test Dependency:**

Production Module

- import TestBlahBlahBlah;

![Diagram](Production Code → Test Code)
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

Agile Development Cycles

- To *prevent* defects,
  the tests must run “quickly enough” to be run often
**Behavior Smell**

**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

---

**Exercise: BS1 - Avoiding Slow Tests**

- **Symptoms:**
  - Your tests are taking 10 minutes to run. Everyone is getting frustrated because it is taking an hour to get the “commit token”.

- **Instructions:**
  - Brainstorm at least 5 ways to make the tests run faster.

- **Discussion Questions:**
  - What might be the root causes of slow test?
  - What can we do to address each possible cause?
Avoiding Slow Tests – Slow SUT

• Run Tests Faster
  – Get faster hardware
    » E.g. Quad-processor test execution box

• Avoid Slow Code
  – Avoid Fixture Persistence
    » Use a Fresh Fixture with Fake Database
  – Avoid slow components
    » Replace with Test Double (or Test Stub)

• Run Fewer Tests
  – Run subsets of tests when possible (e.g. pre-checkin)
  – Run all the tests sometime, somewhere! (e.g. overnight)

Avoiding Slow Tests – Slow Test Code

• Avoid Waits
  – Use Humble Object to avoid Asynchronous Test by testing logic directly

• Test Less Code
  – Reduce Test Overlap

• Set Up Less Fixture
  – Use a Minimal Fixture

• Set Up Fixture Less Often
  – Reuse a Shared Fixture
**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.

**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

**Bad Smell Alert:**
- Erratic Tests
### xUnit Test Patterns and Smells

#### Setting Up the Shared Test Fixture

To share the same fixture *instance* between tests:

- **Prebuilt Fixture**
  - Fixture is built ahead of time and reused by many test runs

- **Lazy Setup**
  - First reference causes it to be initialized
  - How do you know when to clean up?

- **SuiteFixture Setup**
  - Use Static variables to hold the fixture
  - Initialize one before first test; destroy after last

- **Setup Decorator**
  - Define a Test Decorator that implements Test
  - Wrap the test suite with an instance of the decorator

#### Unrepeatable Tests

- Use only when don’t need to clean up the fixture

- Not supported by all frameworks

- Tests that depend on the decorator cannot be run without it.

### xUnit Test Patterns and Smells

#### Behavior Smell

**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.

**Erratic Tests – Non Deterministic Test**

Tests depend on non-deterministic inputs.

**Symptoms:**
- Tests pass at some times; fail at other times
  - Lack of control over time/date when system contains time/date logic (addressed by getting control of indirect input via a stub)
  - Tests use different values in different runs
Erratic Tests – Resource Optimism

Tests depend on non-ubiquitous external resources.

Symptoms:
- Tests pass in some environments; fail in others
  - SUT depends on something in the environment that is not always present.
  - Addressed by creating it during the fixture setup phase

Avoiding Erratic Tests - Fresh Fixture

- What it is:
  - “Brand new” fixture built for each test
  - Tests are completely independent
Fresh Fixture

• Variations:
  – Transient Fresh Fixture
    » Fixture automatically disappears at end of each test
    » e.g. Garbage-collected TearDown
  – Persistent Fresh Fixture
    » Fixture naturally “hangs around” after test
    » Requires extra effort to ensure it is fresh

Persistent Fresh Fixture

Two Options:
1. Rebuild fixture for each test and tear it down
   – When
     » At end of this test (just in case)
     » At start of next test that uses it (just in time)
   – How
     » Hand-coded Tear Down
     » Automated Tear Down
2. Build different fixture for each test
   – Use a Distinct Generated Value for any unique Id’s
   – Makes tear down necessary
Reducing Erratic Tests - Shared Fixture

• Avoid Interactions between Test Runners
  – Give each developer their own Database Sandbox.
    » Avoids Test Run Wars but not Interacting Tests, etc,

• Don’t Change Shared Fixture
  – Immutable Shared Fixture avoids Interacting Tests
  – Create Fresh Fixture for objects to be changed
    » (See Persistent Fresh Fixture)
  – Challenge: What constitutes a “change” to a fixture?
    » Change existing objects / rows -> YES!
    » Add new objects related to existing objects -> SOMETIMES!

Reducing Erratic Tests - Shared Fixture

• Build new Shared Fixture for each run
  – Avoids Unrepeatable Tests
  – When:
    » Lazy Setup
    » Setup Decorator
    » SuiteFixture Setup
Fragile Tests

Causes:

- **Interface Sensitivity**
  - 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.

- **Behavior Sensitivity**
  - Behavior of the SUT changes but it **should not** affect test outcome.
  - Caused by being dependent on too much of the SUT’s behavior.

Fragile Tests (2)

Causes (continued):

- **Data Sensitivity**
  - Alias: Fragile Fixture
  - Tests start failing when a shared fixture is modified.
    - e.g. New records are put into the database

- **Context Sensitivity**
  - Something outside the SUT changes.
    - e.g. System time/date, contents of another application.
**Avoiding Interface Sensitivity**

- **Use Stable Interfaces**
  - Bypass Presentation Layer (UI)
  - Backwards compatibility of changes to used interface
    - e.g. Facade
- **SUT API Encapsulation**
  - Hide non-essential parts of SUT API from Test Methods via
    - Creation Method
    - Finder Method
    - Verification Method

**Avoiding Data/Context Sensitivity**

- **Minimal Fresh Fixture**
  - Use a Fresh Fixture
  - Custom design it for each test.
  - Avoid a Standard Fixture that could become a Fragile Fixture
- **Test Stubs**
  - Replace the need for real fixture by using a Test Stub to provide indirect inputs
**Assertion Roulette**

**Symptom:**
- One or more unit tests are failing in the automated build and you cannot tell why without rerunning the tests in your IDE. When you cannot reproduce the problem in your IDE you have no idea what is going wrong.

**Impact:**
- It takes longer to determine what is wrong with the code.
- Bugs that cannot be reproduced cannot be fixed.

**Root Cause:**
- Missing/Unclear Assertion Messages

**Solution:**
- Use the right Assertion Method.
- Add Assertion Messages to all Assertion Method calls
- Write Diagnostic Custom Assertion

**Diagnostic Custom Assertion**

**Variation of Custom Assertion**

**Compares its inputs in a way that provides useful diagnostic messages.**

**e.g. assertEquals does this:**
- expected <nil> but was <abc>
- strings differ starting at position 247; expected <...abcdefghi..> but was <...abcxyzghi..>
Behavior Smell

Frequent Debugging

- **Symptom:**
  - One or more tests are failing and you cannot tell why without resorting to the debugger. This seems to be happening a lot lately!

- **Impact:**
  - Debugging is a very time-intensive activity.
  - While it may help you find the bug, it won’t keep it from coming back.

- **Root Causes:**
  - Missing Unit Tests
  - Poor Assertion Messages

- **Solution:**
  - Better unit test coverage of the code
  - More/Better Assertion Messages

Outline

- Introduction
- Motivation
- Intro to Smells & Patterns
- Code Smells & Remedies
- Behavior Smells & Remedies
- Project Smells & Remedies *
- 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
More on xUnit Patterns & Smells

• Book: xUnit Test Patterns Refactoring Test Code by: Gerard Meszaros
  – published by Addison Wesley

• Website: http://xunitpatterns.com
  With handy links to purchase

• Onsite Training
  – training at xunitpatterns.com
  – 1-403-827-2967

Thank You!
Gerard

Please fill in your evaluation forms!

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 Vlissides
Exercise CS1 - Result Verification

Situation
You have just inherited maintenance of the Flight Management System. The good news is that there are automated unit tests. The bad news is that most of the tests look something like these tests.

Instructions:
Examine the code in the handout and determine what code smells you are seeing.

Discussion Questions:
• Which Code Smells are we having?
• What are the underlying root causes?
• Which Patterns can we apply to alleviate them?
Test Code

```java
public void testGetFlightsByOriginAirports() throws Exception {
    // create 3 flights from one airport to two anonymous airports and one flight beyond:
    // b->a->c->d
    FlightDto expectedFlight1 = createNewFlightBetweenTwoNewAirports(); // a->b
    FlightDto expectedFlight2 = createNewFlightToNewAirportFrom(expectedFlight1.origAirport); // a->c
    FlightDto expectedFlight3 = createNewFlightToNewAirportFrom(expectedFlight2.destAirport); // c->d

    // Test 1: no flights from the destination of Flight 1:
    // Exercise SUT:
    List flightsFromDest1 = facade.getFlightsByOriginAirport(expectedFlight1.destAirport);
    // Verify outcome:
    assertEquals("Should not be any flights from dest of flight1", 0, flightsFromDest1.size());

    // Test 2: one flight from the destination of Flight 2:
    // Exercise SUT:
    List flightsFromDest2 = facade.getFlightsByOriginAirport(expectedFlight2.destAirport);
    // Verify outcome: # of flights
    if (1 == flightsFromDest2.size()) {
        // Verify attributes of only FlightDto:
        FlightDto onlyFlight = (FlightDto)flightsFromDest2.get(0);
        assertEquals("flight ", expectedFlight1.flightNumber, onlyFlight.flightNumber);
        assertEquals("origAirportId", expectedFlight1.origAirportId, onlyFlight.origAirportId);
        assertEquals("destAirportId", expectedFlight1.destAirportId, onlyFlight.destAirportId);
        assertEquals("origCity", expectedFlight1.origCity, onlyFlight.origCity);
        assertEquals("destCity", expectedFlight1.destCity, onlyFlight.destCity);
        assertEquals("eqiupType", expectedFlight1.equipmentType, onlyFlight.equipmentType);
    } else {
        fail("should be one flight from airport");
    }

    // Test 3: Two flights from the common origination airport:
    // Exercise SUT:
    List flightsFromOrig = facade.getFlightsByOriginAirport(expectedFlight1.origAirport);
    // Verify Outcome:
    // 1st, verify correct number of flights
```
if (flightsFromDest2.size() == 2) {
    // 2nd, verify 1st flight is the one we expected
    // (let's hope they come back in the right order!)
    FlightDto firstFlight = flightsFromDest2.get(0);
    assertEquals("flight ", expectedFlight1.flightNumber, firstFlight.flightNumber);
    assertEquals("origAirportId", expectedFlight1.origAirportId, firstFlight.origAirportId);
    assertEquals("destAirportId", expectedFlight1.destAirportId, firstFlight.destAirportId);
    assertEquals("origCity", expectedFlight1.origCity, firstFlight.origCity);
    assertEquals("destCity", expectedFlight1.destCity, firstFlight.destCity);
    assertEquals("equipType", expectedFlight1.equipmentType, firstFlight.equipmentType);

    // 3rd, verify 2nd flight is one we expected:
    FlightDto secondFlight = flightsFromDest2.get(1);
    assertEquals("flight ", expectedFlight2.flightNumber, secondFlight.flightNumber);
    assertEquals("origAirportId", expectedFlight2.origAirportId, secondFlight.origAirportId);
    assertEquals("destAirportId", expectedFlight2.destAirportId, secondFlight.destAirportId);
    assertEquals("origCity", expectedFlight2.origCity, secondFlight.origCity);
    assertEquals("destCity", expectedFlight2.destCity, secondFlight.destCity);
    assertEquals("equipType", expectedFlight2.equipmentType, secondFlight.equipmentType);
} else {
    fail("should be two flights from airport");
}
Exercise CS2: Fixture Setup

Situation
You have just inherited maintenance of the Flight Management System. The good news is that there are automated unit tests. The bad news is that most of the tests look something like these tests.

Instructions:
Examine the code in the handout and determine what code smells you are seeing.

Discussion Questions:
• Which Code Smells are we having?
• What are the underlying root causes?
• Which Patterns can we apply to alleviate them?
Test Code

```java
public void testGetFlightsByOriginAirports_TwoOutboundFlights() throws Exception {
    // Set up 2 flights from one airport
    // first, the origin airport
    BigDecimal calgaryAirportId = facade.createAirport(
        CALGARY_AIRPORT_CODE, CALGARY_AIRPORT_NAME, CALGARY_CITY);
    // next, the two destination airports
    BigDecimal sanFranAirportId = facade.createAirport(
        SAN_FRAN_AIRPORT_CODE, SAN_FRAN_AIRPORT_NAME, SAN_FRAN_CITY);
    BigDecimal vancouverAirportId = facade.createAirport(
        VANCOUVER_AIRPORT_CODE, VANCOUVER_AIRPORT_NAME, VANCOUVER_CITY);

    // now, the first flight DTO
    FlightDto expectedFlight1 = new FlightDto();
    expectedFlight1.setOriginAirportId(calgaryAirportId);
    expectedFlight1.setOriginCity(CALGARY_CITY);
    expectedFlight1.setDestinationAirportId(sanFranAirportId);
    expectedFlight1.setDestinationCity(SAN_FRAN_CITY);

    // Here's where we actually create the first flight:
    expectedFlight1.setFlightNumber(facade.createFlight(calgaryAirportId, sanFranAirportId));

    // And the second flight DTO:
    FlightDto expectedFlight2 = new FlightDto();
    expectedFlight2.setOriginAirportId(calgaryAirportId);
    expectedFlight2.setOriginCity(CALGARY_CITY);
    expectedFlight2.setDestinationAirportId(vancouverAirportId);
    expectedFlight2.setDestinationCity(VANCOUVER_CITY);

    // Here's where we actually create the second flight:
    expectedFlight2.setFlightNumber(facade.createFlight(calgaryAirportId, vancouverAirportId));

    // Exercise the SUT:
    List flightsFromCalgary = facade.getFlightsByOriginAirport(calgaryAirportId);
    assertEquals("Number of flights originating in Calgary", 2, flightsFromCalgary.size());

    // Verify that flights expectedFlight1 and expectedFlight2 are in the
    // list:
    // etc.
```
Exercise BS 2

Instructions:

Symptoms:
Earlier today, you ran all the tests after making some code changes and the tests ran green. You then went to lunch. When you came back you re-ran the tests “just to make sure” before committing your changes. Now, several tests are failing or erroring.

Discussion Questions:
Based on these symptoms, which Behaviour Smells are we having?
What questions do we need to ask to find out why they are occurring?
What are the underlying root causes?
What can we do about them?

You may also find it useful to ask yourselves these questions:
1. What kind of fixture are these tests using?
2. Which fixture setup pattern is being used?
3. Why is this causing the failures?
Supporting Material:
The following is the test runner and console output from the test runs. For convenience, the developers have decided to use a logging tool to document what is happening in the various parts of the test in the console output.

Suggested Approach
You might find it helpful to draw a sketch of the Airports and Flights as you read the console output.

Console Output (Earlier today->Green)
Testcase Object: testGetFlightsByOriginAirport_OneOutboundFlight
  setUp
  createAirport(YYC)
  createAirport(LAX)
  createAirport(DIA)
  createFlight(YYC-LAX)
  method testGetFlightsByOriginAirport_OneOutboundFlight
  getFlightsByOriginAirport(YYC)
  tearDown
  ****************************************************************************************************************
Testcase Object: testGetFlightsByDestAirport_OneInboundFlight
  setUp
  method testGetFlightsByDestAirport_OneInboundFlight
  getFlightsByDestAirport(LAX)
  tearDown
  ****************************************************************************************************************
Testcase Object: testGetFlightsByOriginAirport_TwoOutboundFlights
  setUp
  method testGetFlightsByOriginAirport_TwoOutboundFlights
  createAirport(YYC)
  createFlight(YYC-DIA)
  getFlightsByOriginAirport(YYC)
  tearDown
XUnit TestRunner Output (Now→2 Failures)
FlightManagementFacadeTest. testGetFlightsByOriginAirport_OneOutboundFlight():
junit.framework.AssertionFailedError: Flights at origin number of flights:
expected:<1> but was:<2>
 at junit.framework.Assert.fail(Assert.java:47)
 at junit.framework.Assert.failNotEquals(Assert.java:282)
 at junit.framework.Assert.assertEquals(Assert.java:64)
 at junit.framework.Assert.assertEquals(Assert.java:201)
FlightManagementFacadeTest. testGetFlightsByOriginAirport_TwoOutboundFlights():
com.clrstream.flightmgnt.FlightManagementError: Airport already exists: ‘DIA’
 at com.clrstream.flightmgnt.FlightManagementFacadeTest.testGetFlightsByOriginAirport_TwoOutboundFlights():

Console Output (Now→Failure)
Testcase Object: testGetFlightsByOriginAirport_OneOutboundFlight
 setUp
 method testGetFlightsByOriginAirport_OneOutboundFlight
 getFlightsByOriginAirport(YYC)
teardown

Testcase Object: testGetFlightsByDestAirport_OneInboundFlight
 setUp
 method testGetFlightsByDestAirport_OneInboundFlight
 getFlightsByDestAirport(LAX)
teardown

Testcase Object: testGetFlightsByOriginAirport_TwoOutboundFlights
 setUp
 method testGetFlightsByOriginAirport_TwoOutboundFlights
 createAirport(DIA)
teardown
Exercise BS 3

Instructions:

Symptoms:
Last week, all the tests ran clean. Since then, 10 new tests have been added (green) but several existing tests are failing.

Discussion Questions:
Based on these symptoms, which Behaviour Smells are we having?
What questions do we need to ask to find out why they are occurring?
What are the underlying root causes?
What can we do about them?

Supporting Material:

The following is the test runner and console output from the test runs. For convenience, the developers have decided to use a logging tool to document the console output with what is happening in the various parts of the test. Extra lines have been added to delineate one test from the other.

Suggested Approach
You might find it helpful to draw a sketch of the Airports and Flights as you read the console output.

XUnit TestRunner Output
FlightManagementFacadeTest.testGetFlightsByOriginAirport_TwoOutboundFlights():
java.util.AssertionFailedError: # of flights at origin 2OF:
expected:<2> but was:<4>
at junit.framework.Assert.fail(Assert.java:47)
at junit.framework.Assert.failNotEquals(Assert.java:282)
at junit.framework.Assert.assertEquals(Assert.java:64)
at junit.framework.Assert.assertEquals(Assert.java:201)
Testcase Object: testGetFlightsByOriginAirport_TwoOutboundFlights
setUp
    setupStandardAirportsAndFlights
    createAirport (NOF)
    createAirport (1OF)
    createAirport (2OF)
    createAirport (MIF)
    createFlight (1OF-MIF)
    createFlight (2OF-MIF)
    createFlight (2OF-1OF)
running testGetFlightsByOriginAirport_TwoOutboundFlights
getFlightsByOriginAirport (2OF)
tearDown
    removeStandardAirportsAndFlights
    removeFlight (1OF-MIF)
    removeFlight (2OF-MIF)
    removeFlight (2OF-1OF)
    removeAirport (NOF)
    removeAirport (1OF)
    removeAirport (2OF)
    removeAirport (MIF)

Testcase Object: testGetFlightsByOriginAirport_TwoOutboundFlights
setUp
    setupStandardAirportsAndFlights
    createAirport (NOF)
    createAirport (1OF)
    createAirport (2OF)
    createAirport (MIF)
    createFlight (1OF-MIF)
    createFlight (2OF-MIF)
    createFlight (2OF-1OF)
addExtraFlights
    createFlight (2OF-MIF)
    createFlight (2OF-1OF)
running testGetFlightsByOriginAirport_TwoOutboundFlights
getFlightsByOriginAirport (2OF)
tearDown
    removeExtraFlights
    removeAirport (2OF-MIF)
    removeAirport (2OF-1OF)
    removeStandardAirportsAndFlights
    removeFlight (1OF-MIF)
    removeFlight (2OF-MIF)
    removeFlight (2OF-1OF)
    removeAirport (NOF)
    removeAirport (1OF)
    removeAirport (2OF)
    removeAirport (MIF)
Exercise BS 4

Symptoms:
You are the first one into the office this morning. You check the builds logs from the overnight build and discover a test failure. When you run the test on your machine it passes. Now what?

Instructions:
Given the following test code and the corresponding TestRunner output, how can you change the test to provide more diagnostic output?

Discussion Questions:
Based on these symptoms, which Behaviour Smells are we having?
What questions do we need to ask to find out why they are occurring?
What are the underlying root causes?
What can we do about them?

Sample Code:
[TestFixture]
public class FlightSchedulerTest
{
    [Test]
    public void TestFlightSchedulerWith2ScheduledFlights()
    {
        Flight flight1 = new Flight("YYC", "LON");
        Flight flight2 = new Flight("LAX", "YYC");
        FlightScheduler sut = new FlightScheduler();
        sut.Schedule(flight1);
        sut.Schedule(flight2);
        Assert.AreEqual(flight2, sut.NextFlight(flight1));
        Assert.AreEqual(flight2, sut.top());
    }
}

Console Output:
TestCase FlightSchedulerTest. testFlightSchedulerWith2ScheduledFlights' failed:
    expected:<flight (LAX->YYC)"">
    but was:NIL
Exercise Solutions

Solutions for the exercises are available at:

http://TutorialSolutions.xUnitPatterns.com