Example

• Test addItemQuantity and removeLineItem methods of Invoice

```
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);
    }
}
```

The Whole Test

```
xUnit Test Patterns and Smells

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 {
            assertEquals("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 {
    fail("Invoice should have exactly one line item");
}

---

**Refactoring**

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

```java
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

```java
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    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");
}
```
### xUnit Test Patterns and Smells

#### Pattern

**Expected Object**

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");
}

---

#### Refactoring

**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");
}
**Refactoring: Introduce Custom Assert**

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

**Conditional Test Logic**

---

**Refactoring: 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 = new LineItem(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 = new LineItem(invoice, product, QUANTITY);
        assertEquals(expectedLineItem, actualLineItem);
    } finally {
        deleteObject(expectedLineItem);
        deleteObject(invoice);
        deleteObject(product);
        deleteObject(customer);
        deleteObject(billingAddress);
        deleteObject(shippingAddress);
    }
}
```

**The Smells Seen Thus Far (1)**

- **Obscure Test**
  - Test is hard to understand.

- **Common Causes:**
  - **Verbose Test**
    - So much test code that it obscures the test intent
  - **Eager Test**
    - Several tests merged into one Test Method
  - **General Fixture**
    - Fixture contains objects irrelevant for this test
  - **Obtuse Assertion**
    - Using the wrong kind of assertion
  - **Hard-Coded Test Data**
    - Lots of “Magic Numbers” or Strings used when creating objects.
    - More likely to result in unrepeateable tests

The Smells Seen Thus Far (2)

• Other Obscure Test Causes:
  – Indirect Testing
    » Interacting with the SUT via other software
    » A cause of Fragile Tests (Behavior Smell)
  – Mystery Guest
    » Lots of “Magic Numbers” or Strings used as keys to database.
    » “Lopsided” feel to tests (either Setup or Verification of outcome is external to test)

The Smells Seen Thus Far (3)

• Conditional Test Logic
  – Tests containing conditional logic (IF statements or loops)
  – Hard to verify correctness. Does it always test the same thing?
  – A cause of Buggy Tests (Project Smell)

• Test Code Duplication
  – Same code sequences appear many times in many tests
  – More code to modify when something changes
  – A cause of Fragile Tests (Behavior Smell)
The Patterns Used So Far

• Expected Objects
  – Use assertEquals on whole objects rather than comparing individual fields

• Guard Assertions
  – Remove conditional logic associated with avoiding assertions when they would fail

• Custom Asserts
  – Remove Test Code Duplication by factoring out common code
  – Remove conditional logic associated with complex verification logic

Replace a series of assertEquals on individual fields:
  – assertEquals (expectedXvalue, actualPoint .getX() );
  – assertEquals (expectedYvalue, actualPoint .getY() );

with a single assertion of the whole object:
  – assertEquals (expectedPoint, actualPoint );

Ways to construct the Expected Object:
  – Point expectedPoint = new Point(17.0, 9.0)
  Or:
  – expectedPoint.setX( 17.0);
  – expectedPoint.sety( 9.0);
Guard Assertion

Conditional Test Logic creates multiple execution paths thru test:

```java
if (actualCollection == null)
    fail("collection is null");
else {
    assertTrue( actualCollection.includes(expectedElement) );
}
```

This makes tests hard to verify.

Better to replace with a Guard Assertion:

```java
assertNotNull( "collection is null", actualCollection);
assertTrue( actualCollection.includes(expectedElement) );
```

Custom Assertion

Remove duplicated assertion logic by creating your own Assertion Methods to:

- **Improve readability**
  - Intent-revealing methods that verify expected outcome

- **Simplify troubleshooting**
  - Make xUnit failure reports easier to understand

- **Define test-specific equality**
  - Ignore "don’t care" fields when comparing objects
  - "Foreign Method" specific to testing

- **Can be defined using Extract Method refactoring.**