Unit Testing Best Practices

Agenda

- Standards vs Principles
- Unit Testing Principles
  - Let's Derive Standards

Principles vs Standards

- Principles
  - a fundamental, primary, or general law or truth from which others are derived: the principles of modern physics.
- Standards
  - Put at its simplest, a standard is an agreed, repeatable way of doing something.
  - Usually derived from Principles

Principles and Standards - Example

- Principle
  - Code should be Understandable
- Corresponding Standards
  - Method Length should be less than XX lines.
  - Cyclomatic Complexity of method should be less than YY.
  - Class should not be more YY lines.
  - No Magic Numbers.

Standards without Principles

- Misinterpretation
  - Method() with complexity 25 might be split into Method1() - 10, Method2() – 10, Method3() - 5
- Some Standards are dependent on the context
  - Code like “public static int ZERO = 0”
- What if the new standards come in?
  - Standards are some body's interpretation of principles and they can change
  - Imagine if rule for complexity is decreased from 10 to 5

Disambiguation – Unit Testing

- *In this presentation, we use the term unit testing to refer to xUnit testing (Junit, Nunit etc) done on individual code units (Methods, Classes).*
- *We do not refer to the screen integration testing performed by a developer on a web application.*

Unit Testing Organization/Attitude

- More important than Code.
  - Lead to Better Design (due to Continuous Refactoring)
- Best written before Code (TDD).
  - *TDD improves Design and Code Quality*
- Separated from Production Code
- Find Defects Early
  - *Continuous Integration*

Unit Testing Principles

1. Easy to understand (Typical test should take no longer than 15 seconds to read.)
2. Test should fail only when there is a problem with production code.
3. Tests should find all problems with production code.
4. Tests should have as minimum duplication as possible.
5. Should run quickly.

Examples we would use

- Amount getClientProductsSum(List<Product>)
  - For a list of products, calculate the sum of product amounts and return total.
  - Throws a DifferentCurrenciesException() if the products have different currencies

Principle 1 : Easy to Understand – Example 1

```
@Test
public void testClientProductSum(){
    List<Product> products = new ArrayList<Product>();
    products.add(new ProductImpl(100, "Product 15", ProductType.BANK_GUARANTEE, new AmountImpl(
            new BigDecimal("5.0"), Currency.EURO)));
    products.add(new ProductImpl(120, "Product 20", ProductType.BANK_GUARANTEE, new AmountImpl(
            new BigDecimal("6.0"), Currency.EURO)));
    Amount temp = null;
    try {
        temp = clientBO.getClientProductsSum(products);
    } catch (DifferentCurrenciesException e) {
        fail();
    }
    assertEquals(Currency.EURO, temp.getCurrency());
    assertEquals(new BigDecimal("11.0"), temp.getValue());
}
```

Principle 1 : Easy to Understand – Example 2

```
@Test
public void testClientProductSum_AllProductsSameCurrency()
throws DifferentCurrenciesException {
    Amount[] amounts = {
        new AmountImpl(new BigDecimal("5.0"), Currency.EURO),
        new AmountImpl(new BigDecimal("6.0"), Currency.EURO) };

    List<Product> products = createProductListWithAmounts(amounts);
    Amount actual = clientBO.getClientProductsSum(products);
    Amount expected = new AmountImpl(new BigDecimal("11.0"), Currency.EURO);
    assertAmount(actual, expected);
}
```

Principle 1 : Easy to Understand

- Name of the Unit Test
  - Should indicate the condition being tested and (if needed) the result
    - *testClientProductSum_AllProductsSameCurrency vs testClientProductSum*
    - *testClientProductSum_DifferentCurrencies_ThrowsException vs testClientProductSum1*
    - *testClientProductSum_NoProducts vs testClientProductSum2*
  - *Keyword test at start of method name is now superfluous. (Junit 4 does not need it.)*
    - *Methods can as well be named clientProductSum_DifferentCurrencies_ThrowsException*

Principle 1 : Easy to Understand

- Highlight values important to the test
  - *Test Setup*

```
List<Product> products = new ArrayList<Product>();

products.add(new ProductImpl(100, "Product 15",ProductType.BANK_GUARANTEE, new
AmountImpl(new BigDecimal("5.0"), Currency.EURO)));

products.add(new ProductImpl(120, "Product 20",ProductType.BANK_GUARANTEE, new
AmountImpl(new BigDecimal("6.0"), Currency.EURO)));
```

COMPARED TO

```
Amount[] amounts = {

    new AmountImpl(new BigDecimal("5.0"), Currency.EURO),

    new AmountImpl(new BigDecimal("6.0"), Currency.EURO) };

    List<Product> products = createProductListWithAmounts(amounts);
```

Principle 1 : Easy to Understand

- Highlight values important to the test
  - *Assertions*

```
Amount expected = new AmountImpl(new BigDecimal("11.0"), Currency.EURO);
assertAmount(expected, actual);
```

COMPARED TO

```
Amount temp = clientBO.getClientProductsSum(products);
assertEquals(Currency.EURO, temp.getCurrency());
assertEquals(new BigDecimal("11.0"), temp.getValue());
```

Principle 1 : Easy to Understand

- One condition per test
  - Results in simple code without if's , for's etc.
  - If a test fails, you would know the exact condition which is failing.
  - Create useful assert methods to test the condition
    - assertAmount(expected, actual);

```
private void assertAmount(Amount expected, Amount actual) {
    assertEquals(expected.getCurrency(), actual.getCurrency());
    assertEquals(expected.getValue(), actual.getValue());
}
```

Principle 1 : Easy to Understand

- No Exception Handling in a test method.

```
public void testClientProductSum_NoProducts() throws DifferentCurrenciesException
```

INSTEAD OF

```
public void testClientProductSum(){
    try {
        temp = clientBO.getClientProductsSum(products);
    } catch (DifferentCurrenciesException e) {
        fail();
    }
}
```

Principle 1 : Easy to Understand

- Use annotated ExceptionHandling to test for exceptions.

```
@Test(expected = DifferentCurrenciesException.class)
public void testClientProductSum_DifferentCurrencies_ThrowsException() throws Exception
{
    CODE THAT THROWS EXCEPTION;
}
```

INSTEAD OF

```
@Test
public void testClientProductSum1() {
    try {
        CODE THAT THROWS EXCEPTION;
        fail("DifferentCurrenciesException is expected");
    } catch (DifferentCurrenciesException e) {
    }}
```

Principle 1 : Easy to Understand - Use new features

- Compare Arrays
  - assertArrayEquals(expectedArray,actualArray)
- Testing Exceptions
  - Annotation (exception = Exception.class)
- Testing Performance
  - Annotation (timeout = 2)
    - 2 milliseconds

Principle 2 : Fail only when there is a defect in CUT (Code Under Test)

- No dependancies between test conditions.
  - Don't assume the order in which tests would run.
- Avoid External Dependancies
  - Avoid depending on (db, external interface,network connection,container).. *Use Stubs/Mocks.*
- Avoid depending on system date and random.
  - Avoid hardcoding of paths (“C:\\TestData\\dataSet1.dat”);//Runs well on my machine..

Principle 3 : Test's should find all defect's in code

- Why else do we write test :)
- Test everything that could possibly break.
  - Test Exceptions.
  - Test Boundary Conditions.
- Use Strong Assertions
  - Do not write “Tests for Coverage”
- Favorite maxim from Junit FAQ
  - *“Test until fear turns to boredom.”*

Principle 3 : Test's should find all defect's in code

- *Junit FAQ : Should we test setters and getters?*

```
becomeTimidAndTestEverything
while writingTheSameThingOverAndOverAgain
    becomeMoreAggressive
    writeFewerTests
    writeTestsForMoreInterestingCases
    if getBurnedByStupidDefect
        feelStupid
        becomeTimidAndTestEverything
    end
End
```

- *Remember this is a infinite loop :)*

Principle 4 : Less Duplication

- No Discussion on this :)

Principle 5 : Test's should run quickly

- To maximize benefits, tests should be run as often as possible. Long running tests stop tests from being run often.
  - Avoid reading from
    - File System or
    - Network
  - A temporary solution might be to *“collect long running tests into a separate test suite” and run it less often*.

Principle 5 : Test's should run quickly

- To maximize benefits, tests should be run as often as possible. Long running tests stop tests from being run often.
  - Avoid reading from
    - File System or
    - Network
  - A temporary solution might be to *“collect long running tests into a separate test suite” and run it less often*.

Result : Tests as Documentation

- Well written tests act as documentation and can be used for discussions with Business as well.
  - Examples
    - testClientProductSum_AllProductsSameCurrency
    - testClientProductSum_DifferentCurrencies_ThrowsException
    - testClientProductSum_NoProducts

Thank You

Questions?