More JUnit

CS 4501 / 6501
Software Testing

[Ammann and Offutt, “Introduction to Software Testing”]
Common methods

- `assertTrue(boolean condition)`
  - Assert that a condition is true

- `assertTrue(String message, boolean condition)`
  - Assert that a condition is true
  - If the assertion is true, the string is ignored. If the assertion is not true, the string is sent to the test engineer.

- `assertEquals(Object expected, Object actual)`
  - Assert that two objects are equal

- `fail(String message)`
  - If a certain situation is expected when a certain section of code is reached, the string is sent to the test engineer.
  - Often used to test exceptional behavior
JUnit – Test Classes

```java
package inclass;

import static org.junit.Assert.*;
import org.junit.Test;

public class ArrayOperationsNumZeroTest {
    @Test
    public void testNumZeroEmptyArray() {
        int x[] = {}; // zero-sized array
        int n = ArrayOperations.numZero(x);
        assertEquals("0 zeros", 0, n);
    }

    @Test
    public void testNumZeroArrayWithNoZeros() {
        int[] x = {1, 2, 3};
        int n = ArrayOperations.numZero(x);
        assertEquals("0 zeros in an array with no zeros", 0, n);
    }
}
```
JUnit – Test Methods

1) Setup test case values

```java
@Test
public void testNumZeroArrayWithNoZeros()
{
    int[] x = {1, 2, 3};
    int n = ArrayOperations.numZero(x);
    assertEquals("0 zeros in an array with no zeros", 0, n);
}
```

2) Execute program under test

3) Assert expected vs. actual test outputs

4) Printed if assert fails
JUnit / xUnit - Conventions

- Group related test methods in a single test class

- The name of test packages/classes/methods should at least transmit:
  - The name of the subject under test (SUT) class
    - TestArrayOperationsNumZero or ArrayOperationsNumZeroTest
  - The name of the method or feature being tested
    - TestArrayOperationsNumZero or ArrayOperationsNumZeroTest
  - The purpose of the test case
    - testNumZeroEmptyArray

- It is common to prefix or suffix test classes with “Test” and prefix test methods with “test”
Exceptions as Expected Results

This pattern is more verbose and unnecessary in this case. It is useful in situations when we wish to perform other assertions beyond the expected exception behavior.
JUnit Test Fixtures

• A test fixture is the state of the test
  • Objects and variables that are used by more than one test
  • Initializations (prefix values)
  • Reset values (postfix values)

• Different tests can use the objects without sharing the state

• Objects used in test fixtures should be declared as instance variables

• Objects should be initialized in a @Before method

• Objects can be deallocated or reset in an @After method
Prefix / Postfix Actions

```
@BeforeClass
public static void globalSetup()
{
    // prefix actions executed once before any test
}

@AfterClass
public static void globalTeardown()
{
    // postfix actions executed once after all tests
}

@Before
public void Setup()
{
    // prefix actions executed before each test
}

@After
public void Teardown()
{
    // postfix actions executed after each test
}
```

*Initialize objects and variables that are used by more than one test*

*Reset objects and variables that are used by more than one test*
Data-Driven Tests

- Sometimes, the same test method needs to be run multiple times, with the only difference being the input values and the expected output.

- **Data-driven** unit tests call a constructor for each collection of test values:
  - Run each set of data values with the same tests.
  - Implement data-driven testing with JUnit `Parameterized` mechanism.
  - `@Parameters` annotation defines collection of data values.
Example: JUnit Data-Driven Unit Test

```java
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;

@RunWith(Parameterized.class)
public class DataDrivenCalculatorTest
{
    public int a, b, sum;

    public DataDrivenCalculatorTest (int a, int b, int sum)
    {
        this.a = a;
        this.b = b;
        this.sum = sum;
    }

    @Parameters
    public static Collection<Object[]> calcValues()
    {
        return Arrays.asList (new Object [][] {{1, 1, 2}, {2, 3, 5}});
    }

    @Test
    public void additionTest()
    {
        assertTrue ("Addition Test", sum == Calculator.add (a,b));
    }
}
```

Returns a collection with 2 arrays of inputs and expected outputs (thus, call the constructor twice)

Test 1
Test values: 1, 1
Expected: 2

Test 2
Test values: 2, 3
Expected: 5

Constructor is called for each triple of values

Data-driven test

Test method uses the instance variables initialized in the constructor call
Wrap-up

• Automate as much as possible to make testing efficient as well as effective

• Test frameworks provide very simply ways to automate our test

• Data-driven testing can suffer from a combinatorial explosion in the number of tests (cross-product of the possible values for each of the parameters in the unit tests)

• Test automation is not “silver bullet” however .. It does not solve the hard problem of testing “What test values to use?”

• This is test design .. The purpose of test criteria

What’s Next?

• Test-Driven Development