Agile Software Development

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

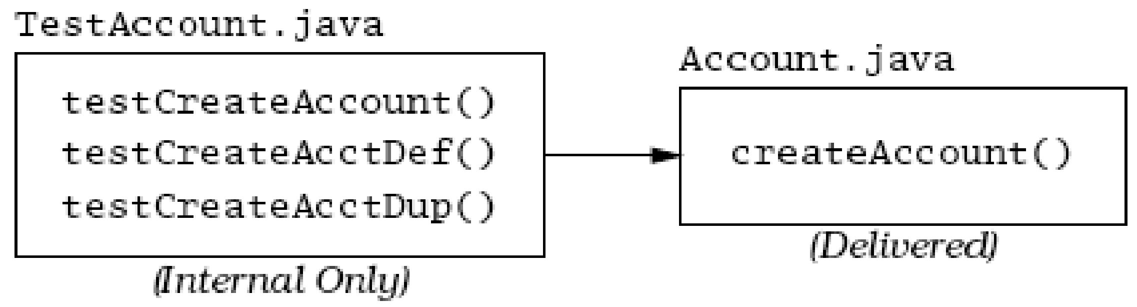
http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

Writing JUnit Tests

Structuring Tests



- Adopt Naming conventions
  - A method named create-Account to be tested, then test method might be named testCreateAccount.
  - The method testCreateAccount will call createAccount with the necessary parameters and verify that createAccount works as advertised.
  - Many test methods that exercise createAccount.
- Distinguish between Testing vs Production Code.
  - The test code is for our internal use only - Customers or end-users will never see it or use it.

Test Code Responsibilities

- 4 steps:
  1. Setup all conditions needed for testing (create any required objects, allocate any needed resources, etc.)
  2. Call the method to be tested
  3. Verify that the method to be tested functioned as expected
  4. Clean up after itself

- Never actually run the production code directly; at least, not the way a user would.
  - Instead, run the test code, which in turn exercises the production code under very carefully controlled conditions.

JUnit Asserts

- Methods that assist in determining whether a method under test is performing correctly or not.
  - Generically called asserts.
  - The developer asserts that some condition is true; that two bits of data are equal, or not equal, or the same, etc...
- Will record failures (when the assertion is false) or errors (when an unexpected exception occurs), and report these through the JUnit classes.
  - The GUI version will show a red bar and supporting details to indicate a failure.
- Asserts are the fundamental building block for unit tests; the JUnit library provides a number of different forms of assert.

assertEquals

- assertEquals([String message], expected, actual)
  - **expected** → a value predicted to be correct (typically hard-coded).
  - **actual** → a value actually produced by the code under test.
  - **message** → an optional and will be reported in the event of a failure.
- Any kind of object may be tested for equality; the appropriate equals method will be used for the comparison (e.g. String.equals()).
- A note of caution: the equals method for native arrays, however, does not compare the contents of the arrays, just the array reference itself.

assertEquals (with Tolerance)

- Computers cannot represent all floating-point numbers exactly, and will usually be off a little bit → a loss of precision.

- Thus using assert to compare floating point numbers (floats or doubles in Java), you should specify one additional piece of information, the **tolerance**.

- assertEquals([String message], expected, actual, **tolerance**)

  - e.g.

    - assertEquals("Should be 3 1/3", 3.33, 10.0/3.0, **0.01**);

assertNull / assertNotNull

- assertNull([String message], java.lang.Object object)
- assertNotNull([String message], java.lang.Object object)
- Asserts that the given object is null (or not null), failing otherwise.

assertTrue / assertFalse

- assertTrue([String message], boolean condition)
- Asserts that the given boolean condition is true, otherwise the test fails.
- If test code is littered with the following:
  - assertTrue(true);
- it suggests that the construct is used to verify some sort of branching or exception logic, it's probably a bad idea and may indicate unnecessarily complex test logic.
- assertFalse([String message], boolean condition)
- Asserts that the given boolean condition is false, otherwise the test fails.

assertSame / assertNotSame

- assertSame([String message], expected, actual)
  - Asserts that **expected** and **actual** refer to the same object, and fails the test if they do not.
- assertNotSame([String message], expected, actual)
  - Asserts that **expected** and **actual** do not refer to the same object, and fails the test if they are the same object.

fail

- fail([String message])
  - Fails the test immediately, with the optional message.
  - Often used to mark sections of code that should not be reached (for instance, after an exception is expected).

Using asserts

- Usually have multiple asserts in a given test method, as you prove various aspects and relationships of the method(s) under test.
- When an assert fails, that test method will be aborted and the remaining assertions in that method will not be executed this time.
- Normally expect that all tests pass all of the time.
- In practice, that means that when a bug introduced, only one or two tests fail.
- Developer should NOT continue to add features when there are failing tests.

JUnit Framework

- The import statement brings in the necessary JUnit methods/annotations.
- Individual tests are marked with the **@Test** annotation against public methods.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class TestClassOne
{

  @Test
  public void testAddition ()
  {
    assertEquals(4, 2 + 2);
  }

  @Test
  public void testSubtraction ()
  {
    assertEquals(0, 2 - 2);
  }
}
```

@Before / @After

- Each test should run independently of every other test; this allows any individual test to be run at any time, in any order.
- This requires ability to reset some parts of the testing environment in between tests, and/or clean up after a test has run.
- **@Before** / **@After** annotations ensure that these methods are called before and after each test is executed.

```
public class TestLargest
{
  private int[] arr;

  @Before
  public void setUp()
  {
    arr = new int[] {8,9,7};
  }

  @After
  public void tearDown()
  {
    arr = null;
  }
}
```

@Before / @After Example

```
public class TestDB extends TestCase
{
  private Connection dbConn;

  @Before
  public void setUp()
  {
    dbConn = new Connection("oracle", 1521,  "fred", "foobar");
    dbConn.connect();
  }

  @After
  public void tearDown()
  {
    dbConn.disconnect();
    dbConn = null;
  }

  @Test
  public void testAccountAccess()  // Uses dbConn
  {
  }

  @Test
  public void testEmployeeAccess()  // Uses dbConn
  {
  }
}
```

@BeforeClass / @AfterClass

- One Time set up for full TestCase.
- Called once before all tests are executed.
- Called once after all tests have executed.
- Does not effect @Before / @After.

```
public class TestDB extends TestCase
{
  private Connection dbConn;

  @Before
  public void setUp()
  {
    dbConn = new Connection("oracle", 1521,  "fred", "foobar");
    dbConn.connect();
  }

  @After
  public void tearDown()
  {
    dbConn.disconnect();
    dbConn = null;
  }

  @BeforeClass
  public static void populateDB()
  {
  }

  @AfterClass
  public static void depopulateDB()
  {
  }
}
```

JUnit Test Composition

- JUnit runs all of the **@Test** annotated methods automatically.
- Individual tests can be removed temporarily via the **@Ignore** annotation.
- **testLongRunner** uses a brute-force algorithm to find the shortest route for the Travelling Salesman Problem (TSP). @Ignore removed it from default tests .....

```
public class TestClassTwo
{
  // This one takes a few hours...
  @Ignore
  @Test
  public void testLongRunner ()
  {
    TSP tsp = new TSP(); // Load with default cities
    assertEquals(2300, tsp.shortestPath(50)); // top 50
  }

  @Test
  public void testShortTest ()
  {
    TSP tsp = new TSP(); // Load with default cities
    assertEquals(140, tsp.shortestPath(5)); // top 5
  }

  @Test
  public void testAnotherShortTest ()
  {
    TSP tsp = new TSP(); // Load with default cities
    assertEquals(586, tsp.shortestPath(10)); // top 10
  }

}
```

Composed Tests

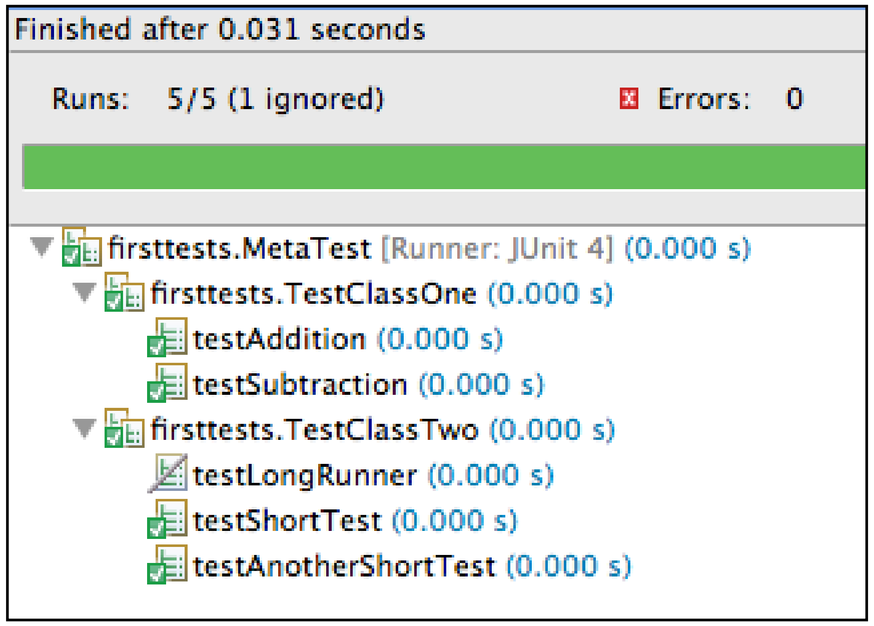
- Higher-level test that is composed of both of two (or more) other test classes.
- The following individual test methods will be run:
  - testAddition()
    from TestClassOne
  - testSubtraction()
    from TestClassOne
  - testShortTest()
    from TestClassTwo
  - testAnotherShortTest()
    from TestClassTwo

```
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;


@RunWith(Suite.class)
@Suite.SuiteClasses({TestClassOne.class,
                     TestClassTwo.class})

public class MetaTest
{

}
```



```
Finished after 0.031 seconds

Runs:   5/5 (1 ignored)                  Errors:   0

▼ firsttests.MetaTest [Runner: JUnit 4] (0.000 s)
  ▼ firsttests.TestClassOne (0.000 s)
      testAddition (0.000 s)
      testSubtraction (0.000 s)
  ▼ firsttests.TestClassTwo (0.000 s)
      testLongRunner (0.000 s)
      testShortTest (0.000 s)
      testAnotherShortTest (0.000 s)
```

Composed Tests

Class Level Annotations:

- **@RunWith**

  JUnit will invoke the annotated class to run the tests, instead of using the runner built into JUnit.

- **@Suite.SuiteClasses**

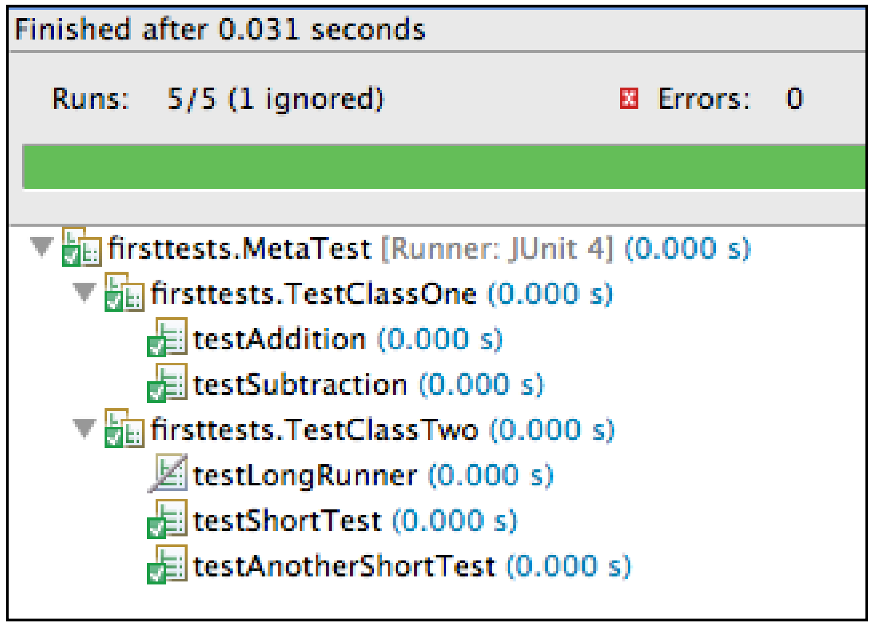
  The SuiteClasses annotation specifies the classes to be executed when a class annotated with @RunWith(Suite.class) is run.

```
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;


@RunWith(Suite.class)
@Suite.SuiteClasses({TestClassOne.class,
                     TestClassTwo.class})

public class MetaTest
{

}
```



```
Finished after 0.031 seconds
Runs:   5/5 (1 ignored)          Errors:   0

▼ firsttests.MetaTest [Runner: JUnit 4] (0.000 s)
  ▼ firsttests.TestClassOne (0.000 s)
      testAddition (0.000 s)
      testSubtraction (0.000 s)
  ▼ firsttests.TestClassTwo (0.000 s)
      testLongRunner (0.000 s)
      testShortTest (0.000 s)
      testAnotherShortTest (0.000 s)
```

Composed Tests: @BeforeClass / @AfterClass

```
@RunWith(Suite.class)
@Suite.SuiteClasses({TestClassOne.class,
                     TestClassTwo.class})

public class MetaTest
{

  @BeforeClass
  public static void initialize()
  {
     System.out.println(“setting up”);
     // …
  }

  @AfterClass
  public static void terminate()
  {
     System.out.println(“tearing down”);
     //...
  }
}
```

```
public class TestClassOne
{
   @Test
   public void test1()
   {
      System.out.println("test1");
      //…
   }
}
```

```
public class TestClassTwo
{
   @Test
   public void test2()
   {
      System.out.println("test2");
      //…
   }
}
```

Output:

```
setting up
test1
test2
tearing down
```

- One time initialization in class MetaTest.
- Then all (non-ignored) tests in TestClassOne and TestClassTwo
- All @Before / @After methods in these classes executed.
- All @BeforeClass / @AfterClass methods also executed.

Good Article on Test Order: https://garygregory.wordpress.com/2011/09/25/understaning-junit-method-order-execution/

JUnit Custom Asserts

- The standard asserts that JUnit provides are usually sufficient for most testing.
- Custom asserts can be introduced by subclassing TestCase and using the subclass for all testing.

```
public class ProjectTest
{

  public void assertEvenDollars (String message, Money amount)
  {
        assertEquals(message, amount.asDouble() -
              (int) amount.asDouble(), 0.0, 0.001);
  }

  public void assertEvenDollars (Money amount)
  {
        assertEvenDollars("", amount);
  }

}
```

JUnit & Exceptions

- There are two kinds of exceptions worth noting:

  Case 1. Expected exceptions resulting from a test

  Case 2. Unexpected exceptions from something that's gone horribly wrong

- For case 2 - JUnit will catch these and provide a complete stack trace.

Expected Exceptions

- For case 1 - sometimes in a test, need to verify that the method under test has actually thrown an exception.

- “expected” annotation parameter declares that the specified exception should have been thrown.

```
@Test
public void testEmpty ()
{
    try
    {
        Largest.largest(new int[] {});
        fail("Should have thrown an exception");
    }
    catch (RuntimeException e)
    {
       assertTrue(true);
    }
}
```

```
@Test (expected = RuntimeException.class)
public void testEmpty ()
{
    Largest.largest(new int[] {});
}
```