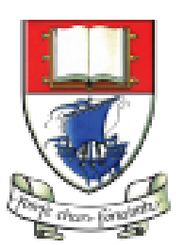
Writing JUnit Tests

Produced by:

Dr. Siobhán Drohan (sdrohan@wit.ie)

Eamonn de Leastar (edeleastar@wit.ie)

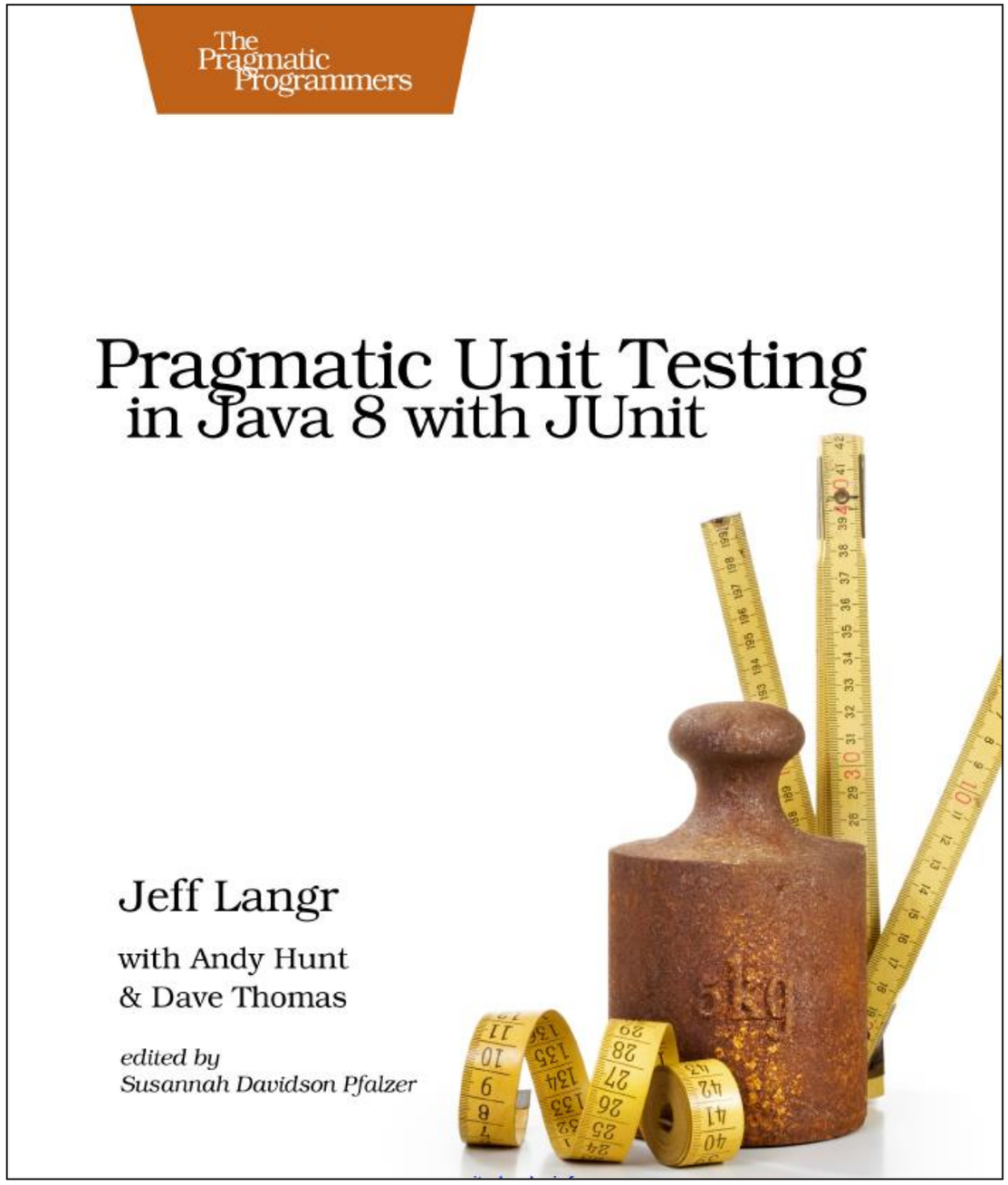


Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

Anatomy of a Unit Test

- Four Phase Test i.e. Setup, Exercise, Verify, Teardown.
- In-Line Setup and Teardown.
- Arrange, Act, Assert.
- Structuring Tests.
- JUnit4 Assertions.

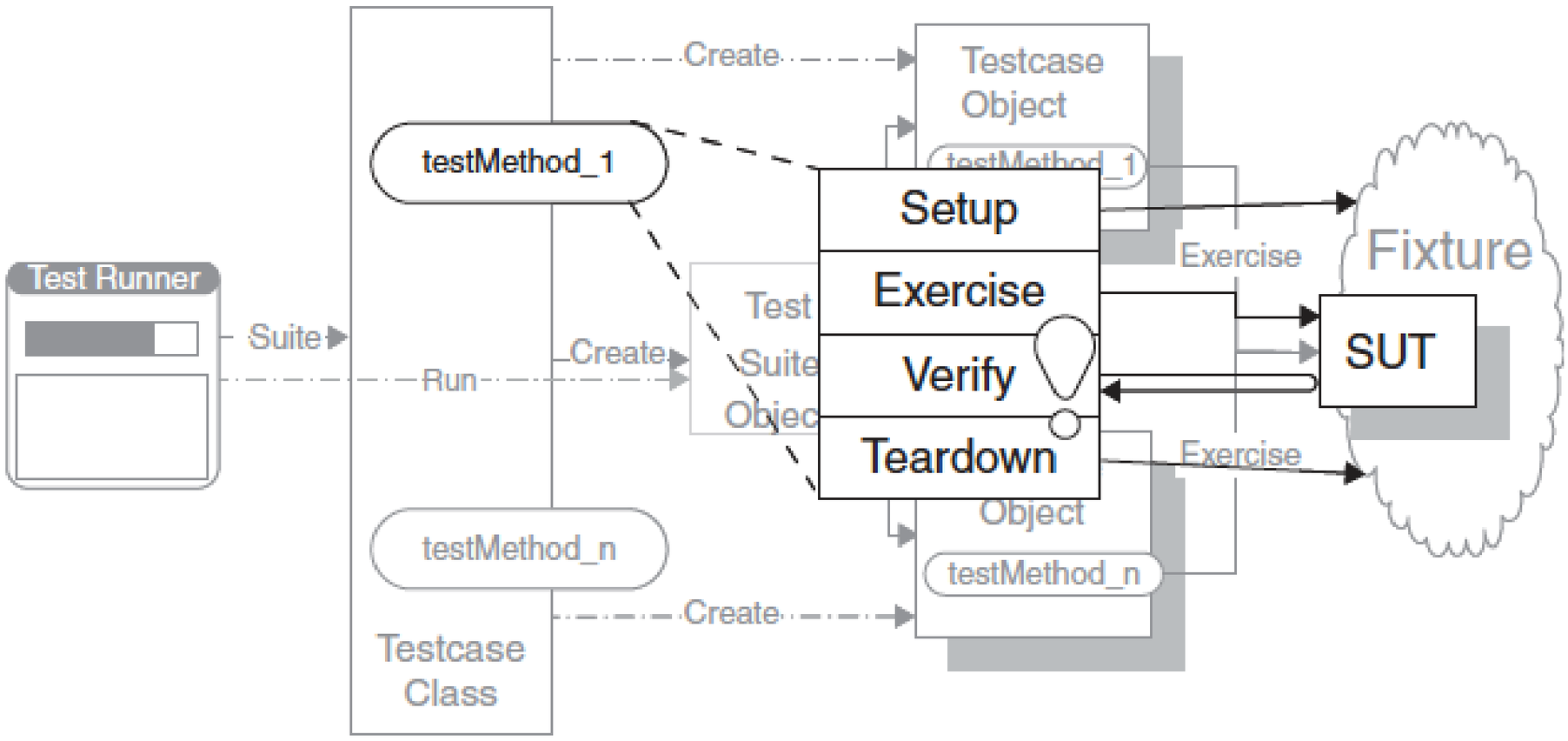


Source Code: https://pragprog.com/titles/utj2/source_code

Four Phase Test

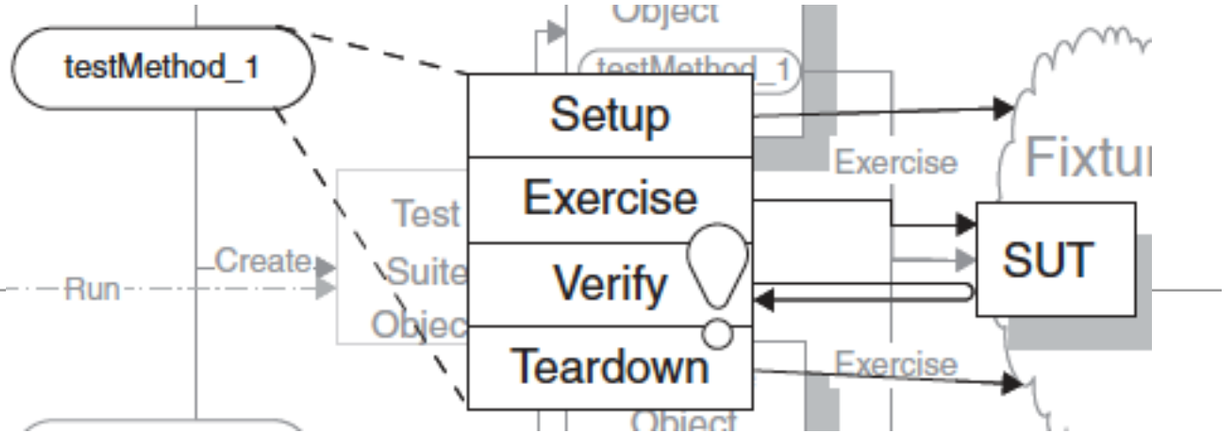
How do we structure our test logic to make what we are testing obvious?

We structure each test with four distinct parts executed in sequence.



SUT = System Under Test

How it works



- SETUP: In the <u>first</u> phase, we set up the test fixture (the “before” picture) that is required for the SUT to exhibit the expected behavior as well as anything you need to put in place to be able to observe the actual outcome.

- EXERCISE: In the <u>second</u> phase, we interact with the SUT.

- VERIFY: In the <u>third</u> phase, we do whatever is necessary to determine whether the expected outcome has been obtained.

- TEARDOWN: In the <u>fourth</u> phase, we tear down the test fixture to put the world back into the state in which we found it.

Four Phase Test: Example

Phase 1
(setup)

Phase 4
(teardown)

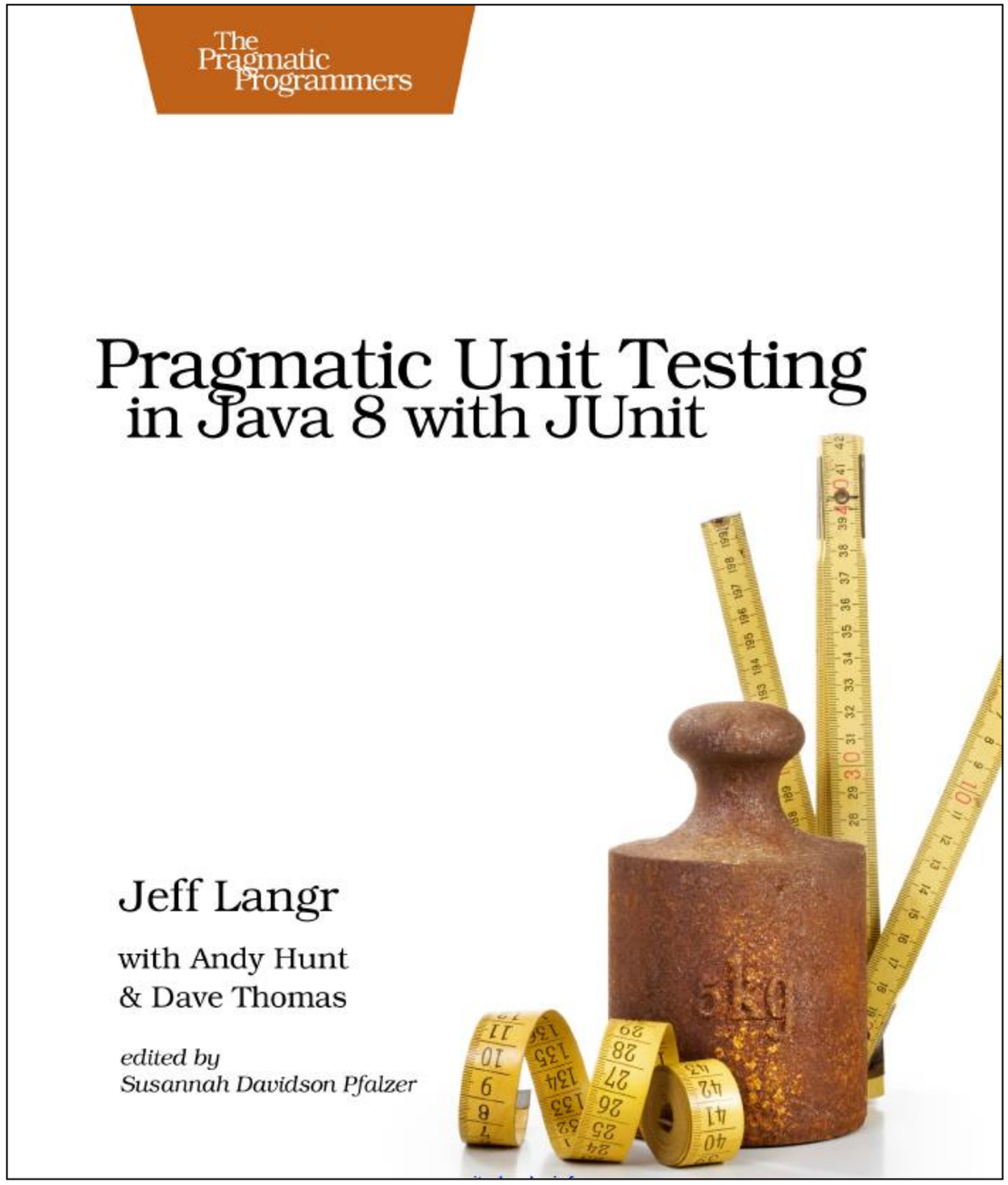
Phase 2 (exercise)
Phase 3 (verify)

PacemakerAPITest.java

```
1  package controllers;
2
3  import static org.junit.Assert.*;
17
18 public class PacemakerAPITest
19 {
20     private PacemakerAPI pacemaker;
21
22     @Before
23     public void setup()
24     {
25         pacemaker = new PacemakerAPI(null);
26         for (User user : users)
27         {
28             pacemaker.createUser(user.firstName, user.lastName, user.email, user.password);
29         }
30     }
31
32     @After
33     public void tearDown()
34     {
35         pacemaker = null;
36     }
37
38     @Test
39     public void testUser()
40     {
41         assertEquals (users.length, pacemaker.getUsers().size());
42         pacemaker.createUser("homer", "simpson", "homer@simpson.com", "secret");
43         assertEquals (users.length+1, pacemaker.getUsers().size());
44         assertEquals (users[0], pacemaker.getUserByEmail(users[0].email));
45     }
46
```

Anatomy of a Unit Test

- Four Phase Test i.e. Setup, Exercise, Verify, Teardown.
- In-Line Setup and Teardown.
- Arrange, Act, Assert.
- Structuring Tests.
- JUnit4 Assertions.



Source Code: https://pragprog.com/titles/utj2/source_code

In-line Setup and Teardown

Phase 1 (setup)

Phase 2(exercise)

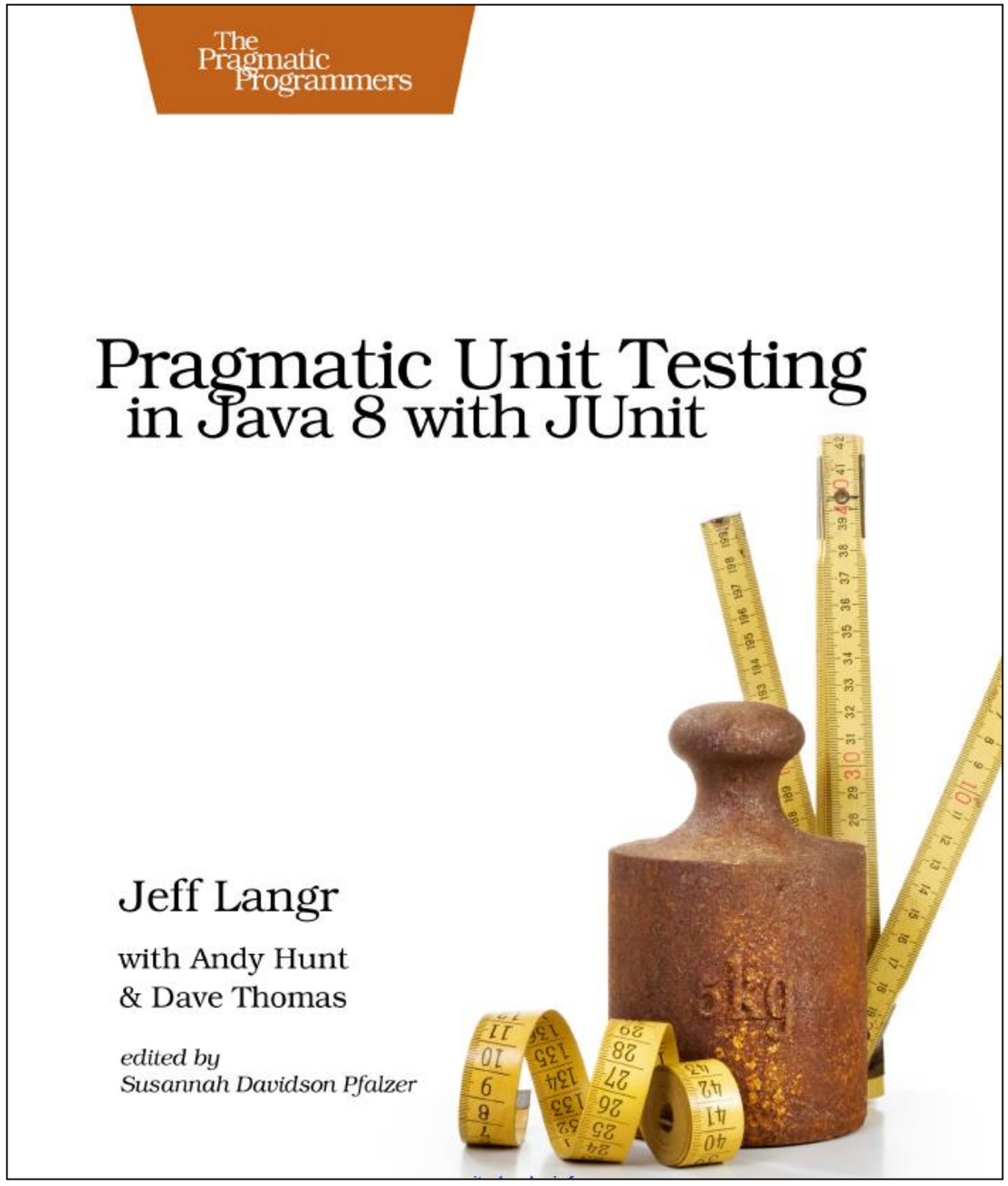
Phase 3 (verify)

Phase 4 (teardown)

```
@Test
public void testXMLSerializer() throws Exception
{
    String datastoreFile = "testdatastore.xml";
    deleteFile (datastoreFile);

    Serializer serializer = new XMLSerializer(new File (datastoreFile));

    pacemaker = new PacemakerAPI(serializer);
    populate(pacemaker);
    pacemaker.store();

    PacemakerAPI pacemaker2 =  new PacemakerAPI(serializer);
    pacemaker2.load();

    assertEquals (pacemaker.getUsers().size(), pacemaker2.getUsers().size());
    for (User user : pacemaker.getUsers())
    {
        Collection<User> users = pacemaker2.getUsers();
        System.out.println("User to search for:");
        System.out.println(user);
        System.out.println("Collection");
        System.out.println(users);
        assertTrue (users.contains(user));
    }
    deleteFile (datastoreFile);
}
```

Anatomy of a Unit Test

- Four Phase Test i.e. Setup, Exercise, Verify, Teardown.
- In-Line Setup and Teardown.
- Arrange, Act, Assert.
- Structuring Tests.
- JUnit4 Assertions.



Source Code: https://pragprog.com/titles/utj2/source_code

Arrange, Act, Assert (AAA)

- An alternative, or a complement to the Four-phases pattern.
- AAA is a pattern of arranging the code within the method itself, something similar to In-Line Setup and Teardown.

Source: http://www.agile-code.com/blog/the-anatomy-of-a-unit-test/

Arrange, Act, Assert

Arrange	To do anything in a test, we first need to *arrange* things with code that sets up the state in a test e.g. creating objects, interacting with them, calling other APIs etc. In some rare cases, we won't arrange anything, because the system is already in the state we need.
Act	After we arrange the test, we *act* on—execute—the code we're trying to verify. Usually this is a call to a single method.
Assert	Finally, we *assert* that we get the expected result. Verify that the exercised code behaved as expected. This can involve inspecting the return value of the exercised code or the new state of any objects involved. It can also involve verifying that interactions between the tested code and other objects took place.
After	*You might need a fourth step…if running the test results in any resources being allocated, ensure that they get cleaned up.*

First, a note on the ***iloveyouboss*** project

- It is a job-search website designed to compete with sites like Indeed and Monster.

- It takes a different approach and attempts to match prospective employees with potential employers, and vice versa, much as a dating site would.

- Employers and employees both create profiles by answering a series of multiple-choice or yes-no questions.

- The site scores profiles based on criteria from the other party and shows the best potential matches from the perspective of both employee and employer.

- A sample question could be “Are you willing to relocate?”.

Arrange, Act, Assert: Basic Example

ScoreCollection.java

```
 2⊕ * Excerpted from "Pragmatic Unit Testing in Java with JUnit",
 9  package iloveyouboss;
10
11  import java.util.*;
12
13  public class ScoreCollection {
14     private List<Scoreable> scores = new ArrayList<>();
15
16⊖    public void add(Scoreable scoreable) {
17        scores.add(scoreable);
18     }
19
20⊖    public int arithmeticMean() {
21        int total = scores.stream().mapToInt(Scoreable::getScore).sum();
22        return total / scores.size();
23     }
24  }
25
26
27
```

A ScoreCollection class accepts a Scoreable instance through its add() method.

A Scoreable object is simply one that can return an int score value.

arithmeticMean() returns the average for a collection of scoreable objects i.e. things that answer with a score.

Nice tutorial on Streams: http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

Arrange, Act, Assert: Basic Example

ScoreCollection.java

```
 2⊕ * Excerpted from "Pragmatic Unit Testing in Java with JUnit",
 9  package iloveyouboss;
10
11  import java.util.*;
12
13  public class ScoreCollection {
14     private List<Scoreable> scores = new ArrayList<>();
15
16⊖    public void add(Scoreable scoreable) {
17        scores.add(scoreable);
18     }
19
20⊖    public int arithmeticMean() {
21        int total = scores.stream().mapToInt(Scoreable::getScore).sum();
22        return total / scores.size();
23     }
24  }
25
26
27
```

—a *test case*—

To test a ScoreCollection object, we can add the numbers 5 and 7 to it and expect that the arithmeticMean() method will return 6.

(5 + 7) / 2 = 6

Arrange, Act, Assert: Basic Example

ScoreCollection.java | *ScoreCollectionTest.java

```
 2⊕ * Excerpted from "Pragmatic Unit Testing in Java with JUnit",..
 9  package iloveyouboss;
10
11⊖ import static org.junit.Assert.*;
12  import static org.hamcrest.CoreMatchers.*;
13  import org.junit.*;
14
15  public class ScoreCollectionTest {
16⊖    @Test
17     public void answersArithmeticMeanOfTwoNumbers() {
18        // Arrange
19        ScoreCollection collection = new ScoreCollection();
20        collection.add(() -> 5);
21        collection.add(() -> 7);
22
23        // Act
24        int actualResult = collection.arithmeticMean();
25
26        // Assert
27        assertThat(actualResult, equalTo(6));
28     }
29  }
30
31
32
```

—a *test case*—

To test a ScoreCollection object, we can add the numbers 5 and 7 to it and expect that the arithmeticMean() method will return 6.

i.e. (5 + 7) / 2 = 6

We will return to this example in a later lecture

Using AAA to complement Four Phase Test

```
public class SomeTestClass
{

	@Before
	public void SetUp()
	{
		//Initialisation of our test
	}

	@Test
	public void Test()
	{
		//Arrange
		// Act
		// Assert
	}

	@After
	public void Teardown()
	{
		//Lets get back to the original state
	}
}
```

Source: http://www.agile-code.com/blog/the-anatomy-of-a-unit-test/

Using AAA to complement Four Phase Test

Phase 1 (setup) / Arrange

Phase 4 (teardown) / After

Act

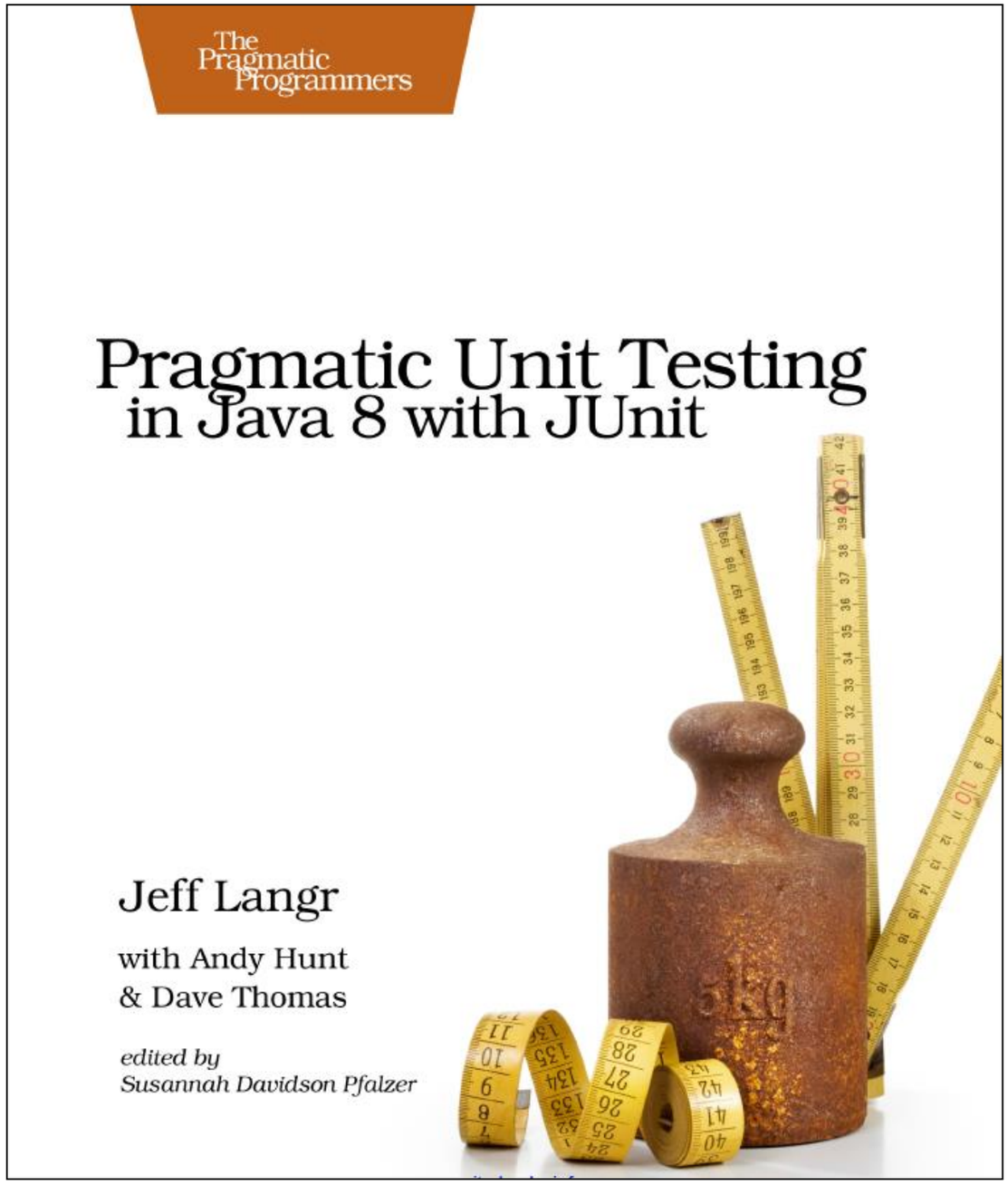
Assert

*PacemakerAPITest.java

```java
package controllers;

import static org.junit.Assert.*;

public class PacemakerAPITest
{
    private PacemakerAPI pacemaker;

    @Before
    public void setup()
    {
        pacemaker = new PacemakerAPI(null);
        for (User user : users)
        {
            pacemaker.createUser(user.firstName, user.lastName, user.email, user.password);
        }
    }

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

    @Test
    public void testUser()
    {
        assertEquals (users.length, pacemaker.getUsers().size());
        pacemaker.createUser("homer", "simpson", "homer@simpson.com", "secret");
        assertEquals (users.length+1, pacemaker.getUsers().size());
        assertEquals (users[0], pacemaker.getUserByEmail(users[0].email));
    }

    @Test
    public void testEquals()
    {
        User homer  = new User ("homer", "simpson", "homer@simpson.com",  "secret");
        User homer2 = new User ("homer", "simpson", "homer@simpson.com",  "secret");
        User bart   = new User ("bart", "simpson", "bartr@simpson.com",  "secret");

        assertEquals(homer, homer);
        assertEquals(homer, homer2);
        assertNotEquals(homer, bart);

        assertSame(homer, homer);
        assertNotSame(homer, homer2);
    }

```

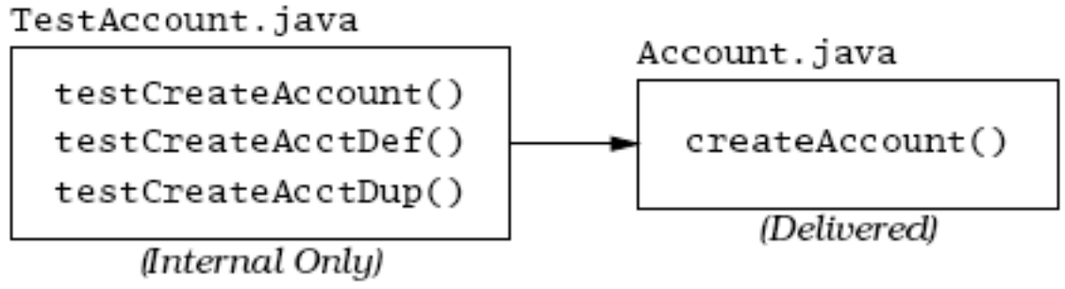
Anatomy of a Unit Test

- Four Phase Test i.e. Setup, Exercise, Verify, Teardown.
- In-Line Setup and Teardown.
- Arrange, Act, Assert.
- Structuring Tests.
- JUnit4 Assertions.



Source Code: https://pragprog.com/titles/utj2/source_code

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 (separate directories in the same project).
  - The test code is for our internal use only - Customers or end-users will never see it or use it.

Naming Individual Tests (1)

- As you move toward more-granular tests, each focused on a distinct behaviour, you have the opportunity to impart more meaning in each of your test names.

- Instead of suggesting what *context* you're going to test, you can suggest what *happens* as a result of invoking some behaviour against a certain context.

- Reasonable test names probably consist of up to seven or so words.

not-so-hot name	cooler, more descriptive name
makeSingleWithdrawal	withdrawalReducesBalanceByWithdrawnAmount
attemptToWithdrawTooMuch	withdrawalOfMoreThanAvailableFundsGeneratesError
multipleDeposits	multipleDepositsIncreaseBalanceBySumOfDeposits

Naming Individual Tests (2)

- The cooler, more descriptive names all follow the form:
  doingSomeOperationGeneratesSomeResult

- You might also use a slightly different form such as:
  someResultOccursUnderSomeCondition

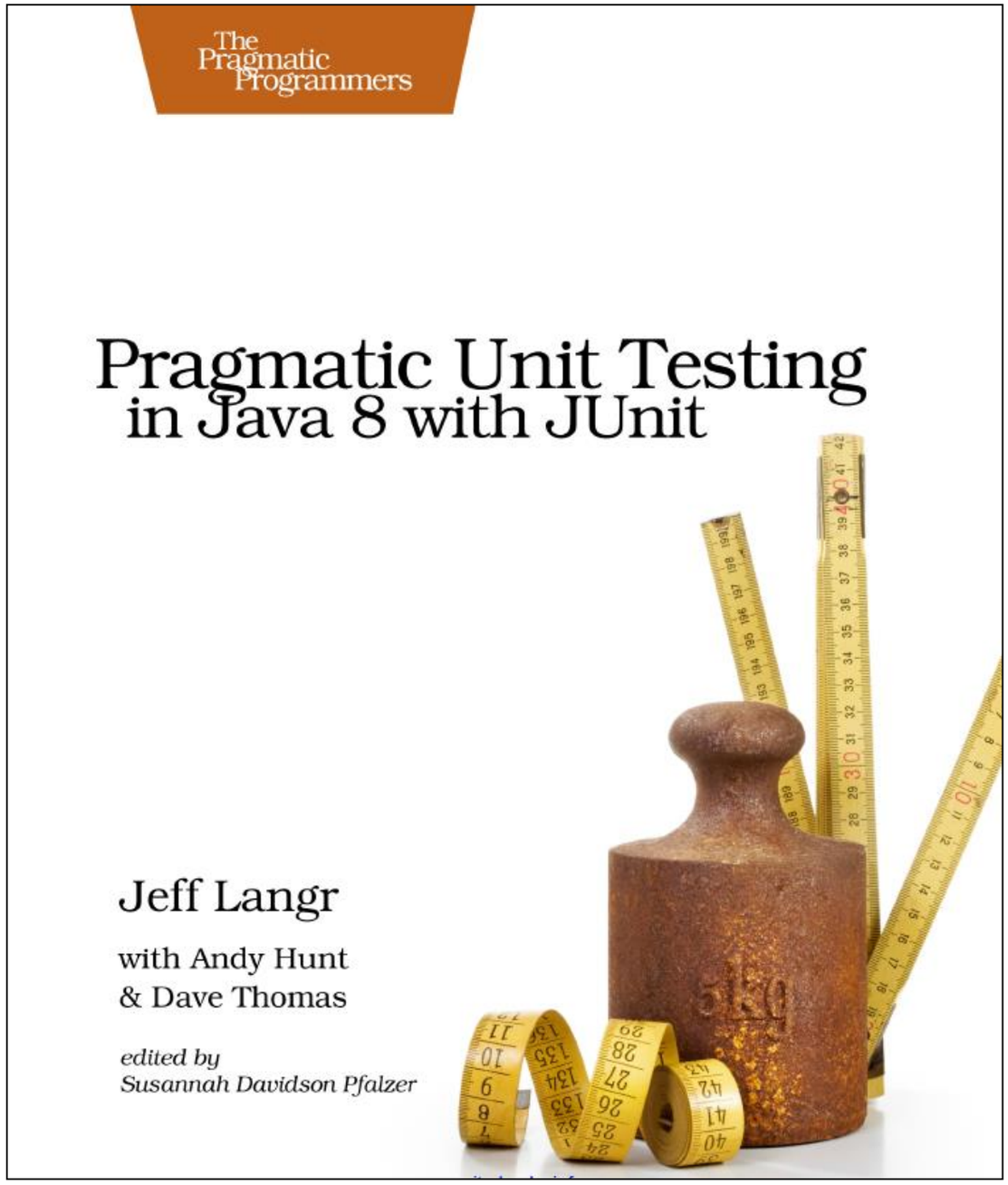
- Or you might decide to go with the *given-when-then* naming pattern (which can be a mouthful):
  givenSomeContextWhenDoingSomeBehaviorThenSomeResultOccurs

- You can usually drop the *givenSomeContext* portion without creating too much additional work for your test reader:
  whenDoingSomeBehaviorThenSomeResultOccurs

- …which is about the same as *doingSomeOperationGeneratesSomeResult*.

Anatomy of a Unit Test

- Four Phase Test i.e. Setup, Exercise, Verify, Teardown.
- In-Line Setup and Teardown.
- Arrange, Act, Assert.
- Structuring Tests.
- JUnit4 Assertions.



Source Code: https://pragprog.com/titles/utj2/source_code

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.

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.

assertThat (using Hamcrest assertion, **equalTo**)

assertThat(*actual*, *matcher*);

- *actual*: value to verify; often a call to the SUT.

- *matcher:* a static method call that allows comparing the results of an expression against an actual value. Matchers can impart greater readability to your tests as they read fairly well left-to-right as a sentence.

```
assertThat(account.getBalance(), equalTo(100));
```

Note: you need to **import** static org.hamcrest.CoreMatchers.*;

assertThat (using Hamcrest assertion, **equalTo**)

assertThat(account.getBalance(), equalTo(100));

- ***equalTo*** uses the equals() method as the basis for comparison.
- Primitive types are autoboxed into instances, so we can compare any type.
- Hamcrest assertions provide a more helpful message when they fail. The prior test expected account.getBalance() to return 100. If it returns 101 instead, you see this:

```
java.lang.AssertionError:
Expected: <100>
     but: was <101>
          at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
```

assertTrue(): when it fails, we get the following stack trace:

```
java.lang.AssertionError
     at org.junit.Assert.fail(Assert.java:86)
```

assertThat (other Hamcrest assertions)

assertThat(account.getName(), **startsWith**(*"xyz"*));

- When the assertThat() call fails, we get the following stack trace:

  java.lang.AssertionError:

  Expected: a string starting with "xyz"

  but: was "an account name“

  at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)

assertThat(account.getName(), **not(equalTo**(*"plunderings"*)));
assertThat(account.getName(), **is(not(nullValue())));**
assertThat(account.getName(), **is(notNullValue()));**

And many more:
http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/CoreMatchers.html

Good comparison of assertThat vs asserts: https://objectpartners.com/2013/09/18/the-benefits-of-using-assertthat-over-other-assert-methods-in-unit-tests/

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.

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.
- You can have multiple methods annotated with @**Before** / @**After** however the order of execution is out of your control; if you require your @Before methods to run in a specific order, resort to just having one method.

```
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.
- Usually used for expensive operations/initialisations e.g. populate a database.

```
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. You can include an explanatory message e.g.:

  @Ignore("takes too long")

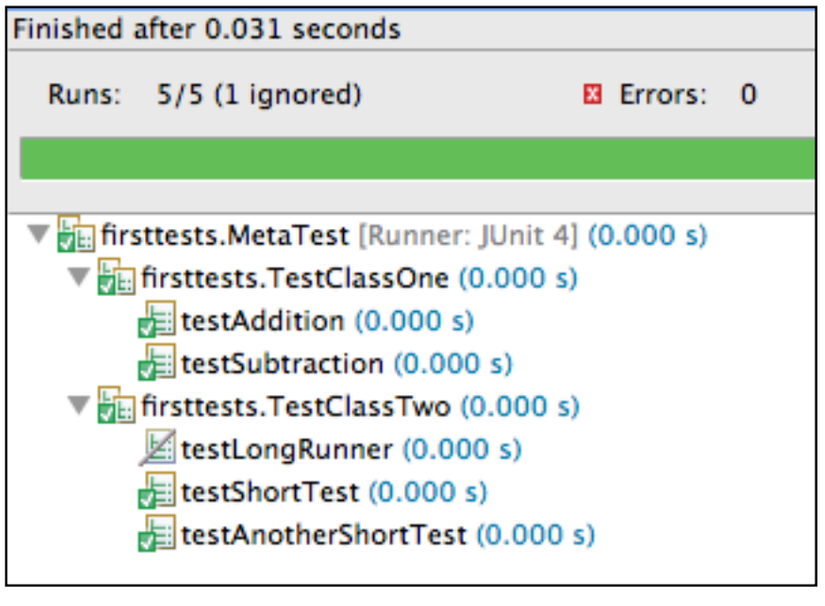
- **testLongRunner** uses a brute-force algorithm to find the shortest route for the Travelling Salesman Problem (TSP). @Ignore removed it from default tests .....

```
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
{


}
```



Composed Tests

Class Level Annotations:

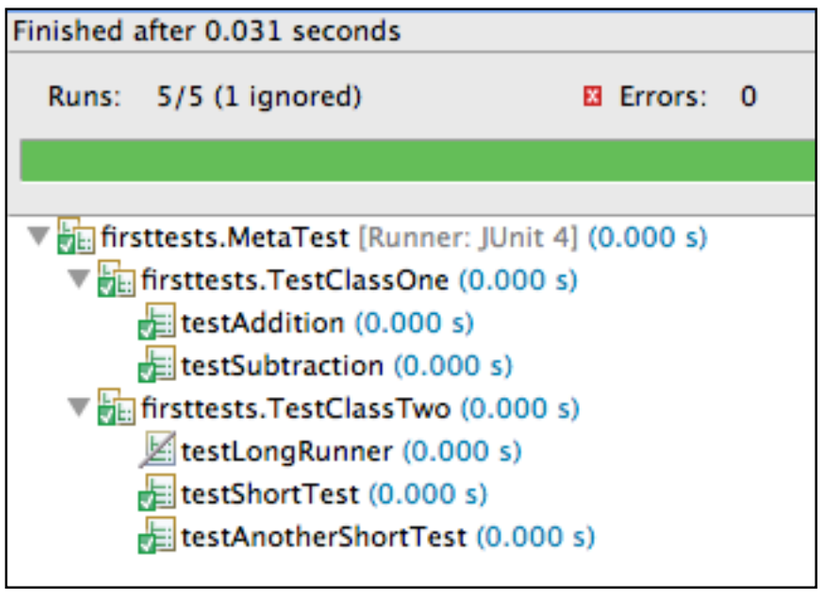
- **@RunWith**

  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
{

}
```



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 & 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.

Old School Approach

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

- *Simple School Approach:* “expected” annotation parameter declares that the specified exception should have been thrown.

Simple School Approach

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

Expected Exceptions Rule – New School Approach for JUnit4

- JUnit allows you to define rules, which can provide greater control over what happens during the flow of test execution.

- Suppose we're designing a test in which we withdraw funds from a new account—that is, one with no money. Withdrawing any money from the account should generate an exception.

- To use the ***ExpectedException*** rule, declare a public instance of ExpectedException in the test class and mark it with ***@Rule***.

New School Approach (JUnit 4)

```
import org.junit.rules.*;
    // ...
    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void exceptionRule() {
        thrown.expect(InsufficientFundsException.class);
        thrown.expectMessage("balance only 0");
        account.withdraw(100);
    }
```

Expected Exceptions Rule – New School Approach for JUnit4

- We tell the ***thrown*** rule instance to expect that an ***InsufficientFundsException*** gets thrown.

- We set another expectation on the ***thrown*** rule…the thrown exception should contain the passed substring.

- Finally, our *act* portion of the test withdraws money which hopefully triggers the exception we expect. JUnit's rule mechanism handles the rest, passing the test if all expectations on the rule were met and failing the test otherwise.

New School Approach (JUnit 4)

```
import org.junit.rules.*;
    // ...
    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void exceptionRule() {
        thrown.expect(InsufficientFundsException.class);
        thrown.expectMessage("balance only 0");
        account.withdraw(100);
    }
```

Testing Exceptions - New School Approach (JUnit 5)

- The @Rule annotation no longer exists in JUnit5; use **assertThrows** instead!

org.junit.jupiter.api

Class Assertions

java.lang.Object
    org.junit.jupiter.api.Assertions

```
@API(value=Maintained)
public final class Assertions
extends java.lang.Object
```

`Assertions` is a collection of utility methods that support asserting conditions in tests.

Unless otherwise noted, a *failed* assertion will throw an `AssertionFailedError` or a subclass thereof.

Since:

`5.0`

See Also:

`AssertionFailedError, Assumptions`

```
static <T extends java.lang.Throwable> assertThrows(java.lang.Class<? extends java.lang.Throwable> expectedType, Executable executable)
T
```

Asserts that execution of the supplied `executable` throws an exception of the `expectedType` and returns the exception.

Testing Exceptions - New School Approach (JUnit 5)

`static <T extends java.lang.Throwable>` **`assertThrows`**`(java.lang.Class<? extends java.lang.Throwable> expectedType,` **`Executable`** `executable)`
`T`

Asserts that execution of the supplied `executable` throws an exception of the `expectedType` and returns the exception.

assertThrows

```
public static <T extends java.lang.Throwable> T assertThrows(java.lang.Class<? extends java.lang.Throwable> expectedType,
                                                             Executable executable)
```

Asserts that execution of the supplied `executable` throws an exception of the `expectedType` and returns the exception.

If no exception is thrown, or if an exception of a different type is thrown, this method will fail.

If you do not want to perform additional checks on the exception instance, simply ignore the return value.

```
@Test
@DisplayName("throws EmptyStackException when popped")
void throwsExceptionWhenPopped()
{
      assertThrows(EmptyStackException.class, () -> stack.pop());
}

@Test
@DisplayName("throws EmptyStackException when peeked")
void throwsExceptionWhenPeeked()
{
      assertThrows(EmptyStackException.class, () -> stack.peek());
}
```

JUnit Testing Advice (so far)

- You should make your tests visually consistent using AAA*(A)*.

- You should keep your tests maintainable by testing behaviour, not methods (i.e. focus on the behaviours of your class and not individual methods).

- Adhere to test naming conventions (and separate folder structures).

- Use @Before and @After for common initialisation and cleanup needs. You can have multiples of these methods.

- Safely ignore tests getting in your way.

Source: http://www.agile-code.com/blog/the-anatomy-of-a-unit-test/