JUnit

COLLABORATORS			
	TITLE : JUnit		*REFERENCE* :
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		May 7, 2010	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 JUnit **1**

1.1 Introduction . . . 1

1.2 Writing JUnit Tests . . . 1

1.2.1 A fist JUnit example . . . 2

1.2.1.1 Writing the unit test class . . . 2

1.2.1.2 Writing test cases . . . 3

1.2.1.3 Making assertions . . . 3

1.2.1.4 Earlier versions of JUnit . . . 3

1.2.2 Unit test classes . . . 3

1.2.2.1 The scope of a unit test . . . 3

1.2.2.2 Lifecycle annotations . . . 4

1.2.3 Test cases . . . 5

1.2.3.1 The scope of a test case . . . 5

1.2.3.2 Raising exceptions . . . 5

1.2.3.2.1 Expecting exceptions . . . 5

1.2.3.3 Asserting performance requirements . . . 6

1.2.4 Assertions . . . 6

1.3 Running JUnit Tests . . . 7

1.3.1 Invoking JUnit through Ant . . . 8

1.3.1.1 Enabling JUnit support . . . 8

1.3.1.2 Automated batch testing . . . 8

1.3.1.2.1 Example implementation . . . 8

2 Index **10**

Chapter 1

JUnit

1.1 Introduction

JUnit (`http://www.junit.org/`) is a framework to test the externally observable behaviour of the components of a system, in order to ensure that the expected behaviour is exhibited under any number of conditions or scenarios.

JUnit encourages *test-driven-development*, a methodology that:

- Increases the trust of the developers and owners in their system,
- Increases developer productivity by setting clear, testable goals at each level of granularity, and
- Improves the modifiability of a system, by reducing the possibility for system changes to have unintended (or unnoticed) side-effects.

JUnit provides the developer with

- A standard way to write *unit tests*, classes that each test some aspect of the behaviour of a system. These tests have a fixed *life cycle*, and a standard mechanism to indicate *test cases* (several of which can be contained within a unit test).
- Infrastructure to make statements (*assertions*) about the expected behaviour of the system. If these assertions are incorrect, the particular test case fails.
- Standard infrastructure to programmatically run tests. Tests are typically executed, however, in the developer's IDE (if it suports JUnit intergation) or, more typically, as an automated task in a project based on the Apache Ant or Maven build systems.

The simple and accessible nature of the JUnit framework make it easy for even inexperienced developers to adopt test-driven development.

1.2 Writing JUnit Tests

A JUnit test consists of a class that contains one or more *test cases* - services that test some aspect of the system.

The following tasks are typically performed for a test case:

- Some form of setup is performed, where the test case creates or looks up the test subject(s)
- The test case makes use of the service(s) of the objects to be tested
- *Assertions* are made as to the state or behaviour of the system. This is typically done in the context of the deliverable of the service (the returned values) but may include expecting failures and performance requirements

1.2.1 A fist JUnit example

Consider a simple utility class that can invert the case of a String:

```
public class StringUtility
{
  /** A service to invert the case of a String - all uppercase
   * characters will be made lowercase, and vice versa.
   * @param s The string to be inverted
   * @return An inverted string, or null if no string was provided
   */
  public static String invertCase( String s)
  {
    if (s == null) return null;

    String invertedString = "";
    for (int i=0; i < s.length(); i++)
    {
      char c = s.charAt(i);
      char ic = Character.isUpperCase(c) ?
          Character.toLowerCase(c) : Character.toUpperCase(c);
      invertedString += ic;
    }
    return invertedString;
  }
}
```

After having written it, we do not know for certain whether it functions as expected. Writing a unit test will allow us to verify (and keep verifying, as the system evolves) this.

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

public class StringUtilityTest
{
  @Test
  public void testInvertCaseNormal()
  {
    String in = "Obi-Wan Kenobi";
    String inverted = StringUtility.invertCase( in );
    assertEquals("oBI-wAN kENOBI", inverted);
  }

  @Test
  public void testInvertCaseNull()
  {
    String inverted = StringUtility.invertCase( null );
    assertNull( inverted );
  }
}
```

1.2.1.1 Writing the unit test class

A unit test is simply any JavaBean class that contains one or more services that have been indicated as *test cases*. It is typical to create one unit test for each class or (more typically) interface that we wish to test, though very complex components under test may have several.

Note
In earlier (pre-4) version of JUnit, test classes had to extend a particular class, `junit.framework.TestCase`.

1.2.1.2 Writing test cases

A test case is any service in your unit test class that

- is `public`
- requires no arguments
- produces no return value
- has been annotated with the `@org.junit.Test` annotation

The test case will typically make use of the services of the object or class under test, and record the output.

Note
In earlier (pre-4) version of JUnit, test cases were not annotated, and had to adhere to a naming convention where the service name started with the word 'test'.

1.2.1.3 Making assertions

The developer makes *assertions* (assumptions) about the output values by making use of the various *assert** services of the `org.junit.Assert` class. If an assertion does not hold (i.e. the values differ from what is expected) the test case will have been recorded as a failure.

1.2.1.4 Earlier versions of JUnit

Note
This example is based on JUnit version 4, which also requires Java version 5 or later. Earlier versions of JUnit do not make use of Java 5 language features, and are implemented using different mechanisms (such as having to extend particular classes, and using particular naming conventions). There are no conceptual differences.

1.2.2 Unit test classes

A JUnit test class contains one or more test cases. It is loaded by the JUnit runtime, and test cases executed, upon request.

```
public class AccountTest
{
  // Test cases go here
  // ...

  private Account testSubject;
}
```

1.2.2.1 The scope of a unit test

Whilst JUnit does not enforce the scope of a unit test, a single test is typically used to test a single class (or interface).

1.2.2.2 Lifecycle annotations

A common requirement in unit tests that contain many test cases is to start with a 'fresh' subject, or set of test data, before executing any particular test case. In order to prevent duplication, we may place such 'setup' instructions in a separate method, and request that it be run before each test case by annotating it with `@org.junit.Before`.

Similarly, we may have to perform cleanup, release resources, or other 'teardown' actions after each test case. A service containing such instructions may be annotated with `@org.junit.After`.

The following skeleton for a test shows the usage of both annotations:

```
import org.junit.*;

public class TestWithBeforeAfter
{
  @Before // Executed before each test case
  public void setUp() throws Exception
  {
    System.out.println("Creating new subject");
    subject = new ClassUnderTest();
  }

  @After //Executed after each test case
  public void tearDown() throws Exception
  {
    System.out.println("Cleaning up");
    // Clean up resources, release connections, etc
    //...
  }

  @Test
  public void testSomething()
  {
    System.out.println("Testing something");
    //...
  }

  @Test
  public void testSomethingElse()
  {
    System.out.println("Testing something else");
    //...
  }

  private ClassUnderTest subject;
}
```

Which, when run, produces the following output:

```
Creating new subject
Testing something
Cleaning up
Creating new subject
Testing something else
Cleaning up
```

Similarly, a service which should be run *only once per unit test, before any test cases are executed* may be annotated with `@org.junit.BeforeClass`, and a service which should be run at *only once, after all test cases have been completed*, may be annotated with `@org.junit.AfterClass`. These are useful to perform global environmental setup, connecting to remote services, etc.

1.2.3 Test cases

A test case is a service within a JUnit test class which tests some aspect of the test subject's behaviour.

```
public class AccountTest
{
  @org.junit.Test
  public void testDebitWithPositiveAmount()
  {
    // Test activities and assertions
    // ...
  }

  @org.junit.Test
  public void testDebitWithNegativeAmount()
  {
    // Test activities and assertions
    // ...
  }

  private Account testSubject;
}
```

1.2.3.1 The scope of a test case

A typical test case should typically test a single service of the subject (and even then, only for a one possible scenario, or set of input data).

Though it is possible to perform a large number of tests within one test case, working at a finer granularity will make it easier to maintain, and might make it easier to identify the cause of failed tests.

1.2.3.2 Raising exceptions

Test cases may declare that they can throw check exceptions. However, should such an exception be thrown, it would result in the test case being marked as a failure (which is probably what the developer wants).

```
public class InventoryTest
{
  @org.junit.Test
  public void testAddProduct() throws InvalidProductException
  {
    // Test activities and assertions
    // (which should not, under normal circumstances, throw
    // an InvalidProductException unless the system is broken)
    // ...
  }
}
```

1.2.3.2.1 Expecting exceptions

In many circumstances, we wish to negatively test some aspect of the system - i.e. the purposeful incorrect usage of the test subject in order to assert that it does indeed refuse to provide the service by throwing an exception.

Using the `expected` parameter to the `@org.junit.Test` annotation asserts that a test should be marked as failed *unless the expected exception was thrown.*

```
public class TelephoneTest
{
```

```
  @org.junit.Test( expected=LineBusyException.class )
  public void testLineBusyExceptionOnDuplicateCall()
  throws LineBusyException
  {
    // Make a call
    subject.initiateCall( "082-555-5555" );

    // Try to make a suplicate call (should fail)
    subject.initiateCall( "082-555-5555" );
  }

  private Telephone subject;
}
```

Note
In many cases, it is not a good idea to enforce (or assert) that a service *must* throw an exception if a particular precondition is not met. In most cases, a service provider *may* refuse to provide the service if the precondition is not met. A service provider may still find a way to realise the requested service (even in the face of a missing precondition) and a unit test should not prevent such innovation.

1.2.3.3 Asserting performance requirements

JUnit provides a simple way to assert the performance requirements of a service, by allowing us to specify the maximum amount of time a test case is allowed to consume before being considered a failure. This is specified with the `timeout` parameter (in miliseconds) to `@org.junit.Test`.

```
public class LibraryTest
{
  /* Not allowed to take more than 1s */
  @org.junit.Test( timeout=1000 )
  public void testFindBooksByName()
  {
    List<Book> books = subject.findBooksByName("Parry Hotter");

    // Further assertions
    // ...
  }

  private LibraryBean subject;
}
```

1.2.4 Assertions

An assertions is a statement which, if untrue, indicates a test failure. JUnit provides a range of assertion tests as static services provided by the `org.junit.Assert` class. These are typically *statically imported* to improve the readability of tests.

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

public class CalculatorTest
{
  @Test(timeout=50)
  public void testAdditionSimple()
  {
    assertEquals( 5, calc.evaluate("3 + 2") );
  }
```

```
  @Test(timeout=50)
  public void testRandom()
  {
    // Test that random floating point number is
    // always between 0.0 and 1.0
    assertTrue( calc.randomNumber() > 0.0 );
    assertTrue( calc.randomNumber() < 1.0 );
  }

  @Before
  public void createCalculator()
  {
    // Perform each test case with a new calculator instance
    calc = new BasicCalculator();
  }

  // The test subject
  private Calculator calc;
}
```

The `org.junit.Assert` class contains a large number of assert services (a subset of which are shown below), such as to assert that

- **a boolean statement is true,** `assertTrue( boolean condition )`
- **a boolean statement is not true,** `assertFalse( boolean condition )`
- **an object is null,** `assertNull( Object object )`
- **an object is not null,** `assertNotNull( Object object )`
- **two objects are logically equal,** `assertEquals( Object expected, Object actual )`
- **two double numbers are equal to within the given precision (delta),** `assertEquals( double expected, double actual, double delta)`

etc.

Services are also provided to immediately fail the currently executing test case, `fail()` and `fail( String message )`.

Note

The behaviour of JUnit assertions are not, in principle, inconsistent with the assertions provided in the Java language by the `assert` keyword: both evaluate a statement to test the developer's assumptions regarding the state of the system.
JUnit, however, is about making statements regarding the externally visible state (behaviour) of a system, whilst java language assertions test the internal integrity of components (and *fail fast* when the system is in an inconsistent state - if assertions are enabled for the current Java VM)

1.3 Running JUnit Tests

There are, in general, three approaches to executing your unit tests: As a built-in feature of your IDE or development environment, as an Ant task, or programmatically via Java method calls.

1.3.1 Invoking JUnit through Ant

The preferred mechanism for invoking JUnit tests are to automate them with Ant (or the higher-level Maven tool, if desired). This enables the ability to

- discover and run all unit tests for a project with one request
- file the reports in a format (such as text, or xml) for later inspection

Apache Ant comes pre-packaged with support for JUnit testing through the `<junit>` task. It supports a large number of options, such as

- running a single or a batch test
- whether test execution should halt on failure (thus indicating an overall failure from an ant perspective) or continue even if one or more tests failed
- configurable output format for test reports, and the ability to file them in a directory for later inspection

1.3.1.1 Enabling JUnit support

Even though ant supports JUnit, it does not, by default, ship the `junit-*.jar` library required to actually write and execute tests. Support can be installed by obtaining the file (from `http://www.junit.org/` and by either *placing this file in the `lib` directory of your ant installation*, or by *packaging the file with your project, and making it available on a class path when ant is used to invoke JUnit*.

1.3.1.2 Automated batch testing

In practise, the most efficient testing strategy from an ant developer's point of view is to add an ant `target` (use-case) for batch testing all JUnit tests in the project. Ass test cases are added, they are automatically identified and run without any further additions to the ant build script.

The best mechanism to distinguish tests from other code is to adhere to a naming convention for the test cases, and the packages they are in. This allows the other build and assembly targets to easily omit packaging unit tests for deployment on a production environment.

Though any convention would do, the following one is sensible:

- Each test case is a class ending with the name `Test`. For example: `AccountTest`, `ConnectionTest` or `PedagogicalDependencyResolverTest`
- The test cases (together with all resources required for testing, such as sample data files) for each Java package are packaged in a sub-package called `tests`. For example, the test case for class or interface `org.foo.bar.X` could be packaged as `org.foo.bar.tests.XTest`

1.3.1.2.1 Example implementation

The following ant target could be re-used across projects to perform batch testing. The properties controlling the pattern to find unit tests, as well as the formatting and test reports directory, are defined in an external file called `build.properties`:

```
# Pattern to select test cases
tests.pattern : **/tests/*Test.*

# Directory to write test reports
tests.reports.dir : testReports

# Output format (xml | plain | brief)
tests.reports.format : brief
```

The ant build script (`build.xml`) could import these properties, and define a target to perform testing:

```
<project name="someProject">

  <!-- Import external properties -->
  <property file="build.properties"/>

  <!-- Class path for your project, possibly including junit-*.jar
       (if not globally installed), as well as the rest of the
       code and dependencies of the project -->
  <path id="classpath">
    ...
  </path>

  <target name="build" description="Compiles the system">
    ...
  </target>

  <target name="test" depends="build" description="Runs all unit tests">
      <!-- Create and/or clean test reports directory -->
      <mkdir dir="${tests.reports.dir}"/>
      <delete>
          <fileset dir="${tests.reports.dir}" includes="**/*"/>
      </delete>
      <!-- Find and run all JUnit tests that match the test pattern -->
      <junit printsummary="yes" haltonfailure="yes" fork="true">
          <!-- Refer to project class path -->
          <classpath refid="classpath"/>
          <!-- Perform a batch test -->
          <batchtest fork="yes" todir="${tests.reports.dir}">
              <formatter type="${tests.reports.format}"/>
              <fileset dir="build">
                  <include name="${tests.pattern}"/>
              </fileset>
          </batchtest>
      </junit>
  </target>

</project>
```

Executing

```
ant test
```

will discover and run all unit tests. If any tests fail, the test will be halted, and an error report placed in the test reports directory.

Chapter 2

Index

J
JUnit, 1
    assertions, 6
    test case, 5
    unit test, 3