JUnit:
Testing code as you write it

Prepared By:

Priya Sharma Trivedi

Introduction

- What is JUnit?
- Downloading and installing JUnit
- Writing JUnit tests
- Creating a test suite
- Output of a JUnit test
- Textual and graphical Test Runners
- Demo
- Conclusions

What is JUnit?

- JUnit is an **open source framework** that has been designed for the purpose of **writing and running tests** in the Java programming language.
- Originally written by **Erich Gamma and Kent Beck**. There are many ways to write test cases.

Regression-Testing Framework

- JUnit is a regression-testing framework that developers can use to write unit tests as they develop systems.
- This framework creates a relationship between development and testing. You start coding according to the specification & need after that use the JUnit test runners to verify how much it deviates from the intended goal.
- This really helps to develop test suites that can be run any time when you make any changes in your code.
- This all process will make you sure that the modifications in the code will not break your system without your knowledge

Why Use JUnit?

- JUnit provides also a graphical user interface (**GUI**) which makes it possible to write and test source code quickly and easily.
- To write and run repeatable tests.
- Programmers normally write their code, then write tests for it. Better to write tests while writing code.
- JUnit provides a framework to keep tests small and simple to test specific areas of code.
- JUnit shows test progress in a bar that is **green** if testing is going fine and it turns **red** when a test fails.
- We can run **multiple tests** concurrently. The simplicity of JUnit makes it possible for the software developer to easily correct bugs as they are found.

JUnit helps the programmer:

- Define and execute tests and test suites
- Formalize requirements and clarify architecture
- Write and debug code
- Integrate code and always be ready to release a working version
- Typically, in a unit testing, we start testing after completing a module but JUnit helps us to code and test both during the development. So it sets more focus on testing the fundamental building blocks of a system i.e. one block at a time rather than module level functional testing.

Download & Installation of JUnit

- Eclipse has JUnit already included, therefore if you intend to use JUnit within Eclipse you don't have to download anything.
- **Downloading :**
  You can download JUnit 4.3.1 from **http://www.junit.org/index.htm**in the zipped format.
- Unzip the JUnit4.3.1.zip file.
- Add **junit-4.3.1.jar** to the CLASSPATH.

  or
- we can create a bat file "**setup.bat**" in which we can write "**set CLASSPATH=.;%CLASSPATH%;junit-4.3.1.jar;**". So whenever we need to use JUnit in our project then we can run this setup.bat file to set the classpath.

Guidelines for using JUnit

- Make sure to test before and after integration with other modules.
- Assume a feature DOESN’T work until you have proven that it does by testing everything that could possibly break.
- Use informative test names.
- Try to test only one thing per method to make it easier to find where error occurred.
- Write tests immediately after you write the function, when it’s fresh in your mind.

What JUnit does

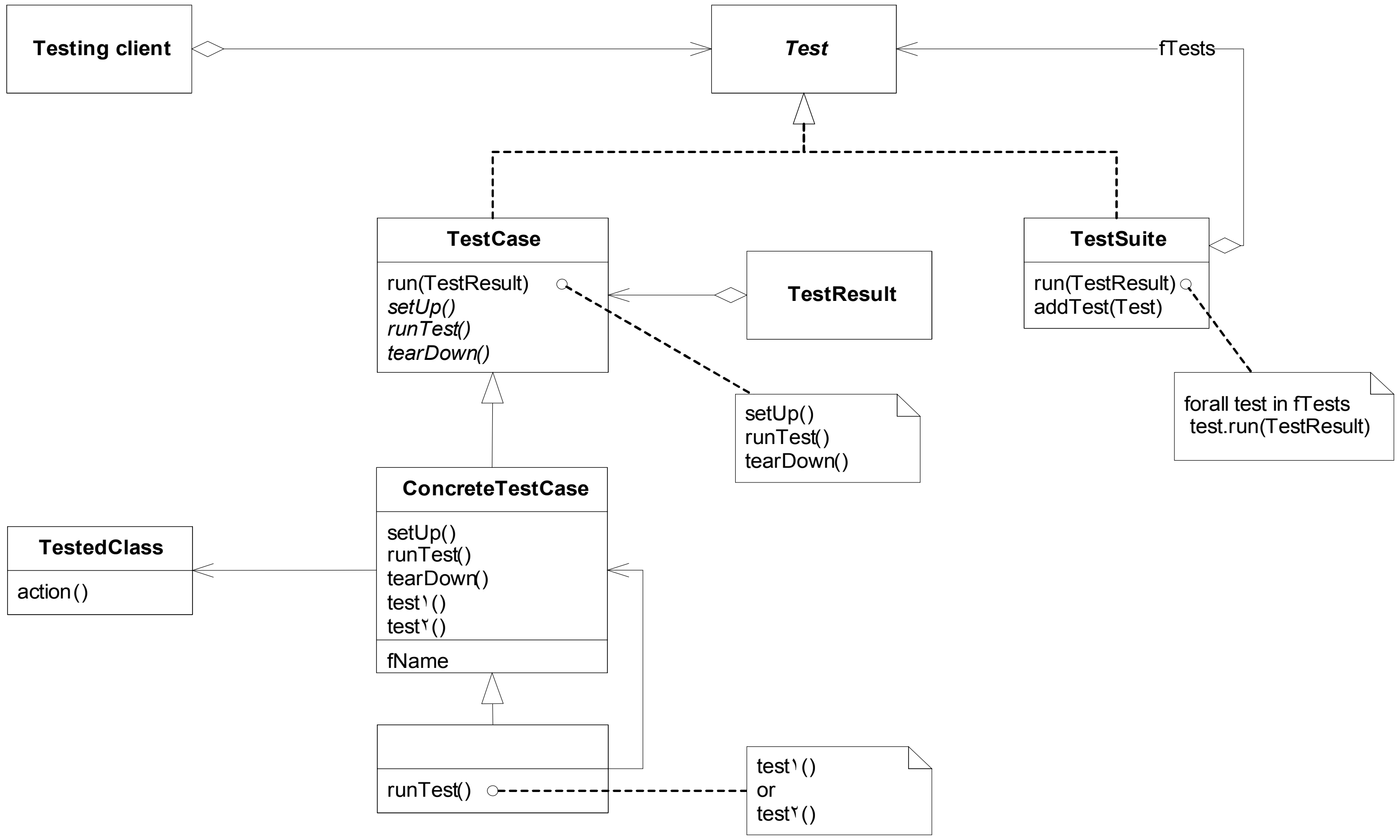
- JUnit runs a suite of tests and reports results.
- For *each* test in the test suite:

  JUnit calls setUp()
    - This method should create any objects you may need for testing

  JUnit calls tearDown()
    - This method should remove any objects you created

  JUnit calls *one* test method
    - The test method may comprise multiple test cases; that is, it may make multiple calls to the method you are testing
    - In fact, since it's your code, the test method can do anything you want
    - The setUp() method ensures you *entered* the test method with a virgin set of objects; what you do with them is up to you

JUnit framework



Creating a test class in JUnit

- Define a subclass of TestCase
- Override the setUp() method to initialize object(s) under test.
- Override the tearDown() method to release object(s) under test.
- Define one or more public testXXX() methods that exercise the object(s) under test and assert expected results.
- Define a static suite() factory method that creates a TestSuite containing all the testXXX() methods of the TestCase.
- Optionally define a main() method that runs the TestCase in batch mode.

For a simple test case, you follow 3 simple steps:

- **Create a subclass of junit.framework.TestCase.**
- **Provide a constructor, accepting a single String name parameter, which calls super (name).**
- **Implement one or more no-argument void methods prefixed by the word test.**

Coding Convention for JUnit Class

- Name of the test class must end with "Test".
- Name of the method must begin with "test".
- Return type of a test method must be void.
- Test method must not throw any exception.
- Test method must not have any parameter.

An example

```
package org.example.antbook.junit;

import junit.framework.TestCase;


public class TestCalculator extends TestCase {
        public TestCalculator(String name) {
                super(name);
        }
        Calculator calculator;
              /** @throws java.lang.Exception  */
        protected void setUp() throws Exception {
                calculator = new Calculator();
        }
                /** @throws java.lang.Exception */
        protected void tearDown() throws Exception {
        }
        public void testAdd() {
                // int sum = calculator.add(6, 7);
                // assertEquals(sum, 15);
                assertEquals(15, calculator.add(8, 7));
        }
}
```

Fixtures

- A fixture is just a some code you want run before every test
- You get a fixture by overriding the method

```
protected void setUp() { …}
```

- The general rule for running a test is:

```
protected void runTest() {
      setUp();  <run the test> tearDown();
    }
```

  so we can override setUp and/or tearDown, and that code will be run prior to or after every test case

Implementing setUp() & tearDown() method

- Override setUp() to initialize the variables, and objects
- Since setUp() is your code, you can modify it any way you like (such as creating new objects in it)
- Reduces the duplication of code

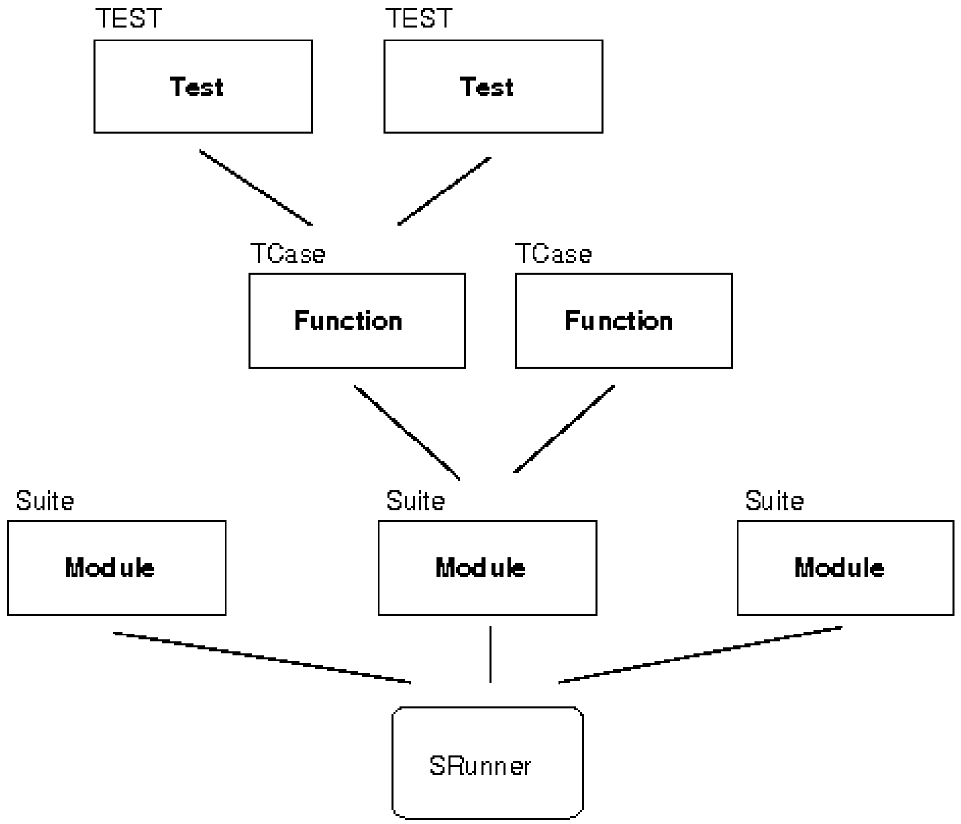
- In most cases, the tearDown() method doesn't need to do anything.
- The next time you run setUp(), your objects will be replaced, and the old objects will be available for garbage collection.
- Like the finally clause in a try-catch-finally statement, tearDown() is where you would release system resources (such as streams)

The structure of a test method

- A test method doesn’t return a result
- If the tests run correctly, a test method does nothing
- If a test fails, it throws an `AssertionFailedError`.
- The JUnit framework catches the error and deals with it; you don’t have to do anything

Test suites

- In practice, you want to run a group of related tests (e.g. all the tests for a class).
- To do so, group your test methods in a class which extends TestCase.
- Running suites we will see in examples.



Writing Tests for JUnit

- Need to use the methods of the junit.framework.assert class
  - javadoc gives a complete description of its capabilities
- Each test method checks a condition (assertion) and reports back to the test runner whether the test failed or succeeded
- The test runner uses the result to report to the user (in command line mode) or update the display (in an IDE)
- All of the methods return void
- A few representative methods of junit.framework.assert
  - assertTrue(boolean)
  - assertTrue(String, boolean)
  - assertEquals(Object, Object)
  - assertNull(Object)
  - Fail(String)

All the above may take an optional String message as the first argument, for example,
static void assertTrue(String message, boolean test)

Sample Assertions

- static void assertEquals (boolean expected, boolean actual)
  - Asserts that two booleans are equal
- static void assertEquals (byte expected, byte actual)
  - Asserts that two bytes are equal
- static void assertEquals (char expected, char actual)
  - Asserts that two chars are equal
- static void assertEquals (double expected, double actual, double delta)
  - Asserts that two doubles are equal concerning a delta
- static void assertEquals (float expected, float actual, float delta)
  - Asserts that two floats are equal concerning a delta
- static void assertEquals (int expected, int actual)
  - Asserts that two ints are equal

- For a complete list, see
- – *http://junit.sourceforge.net/javadoc/org/junit/Assert.html*

Organize The Tests

- Create test cases in the same package as the code under test.
- For each Java package in your application, define a TestSuite class that contains all the tests for validating the code in the package.
- Define similar TestSuite classes that create higher-level and lower-level test suites in the other packages (and sub-packages) of the application.
- Make sure your build process includes the compilation of all tests.

Example: Counter class

- For the sake of example, we will create and test a trivial “counter” class
  - The constructor will create a counter and set it to zero
  - The increment method will add one to the counter and return the new value
  - The decrement method will subtract one from the counter and return the new value
- We write the test methods before we write the code
  - This has the advantages described earlier
  - Depending on the JUnit tool we use, we *may* have to create the class first, and we *may* have to populate it with stubs (methods with empty bodies)
- Don’t be alarmed if, in this simple example, the JUnit tests are more code than the class itself

The Counter class itself

```
public class Counter {
      int count = 0;
      public int increment() {
         return ++count;
      }
      public int decrement() {
         return --count;
      }
     public int getCount() {
        return count;
     }
}
```

JUnit tests for Counter

```
public class CounterTest extends junit.framework.TestCase {
    Counter counter1;
      public CounterTest() { }   // default constructor
        protected void setUp() {   // creates a (simple) test fixture
            counter1 = new Counter();
        }
        protected void tearDown() { } // no resources to release

        public void testIncrement() {
            assertTrue(counter1.increment() == 1);
            assertTrue(counter1.increment() == 2);
         }

         public void testDecrement() {
            assertTrue(counter1.decrement() == -1);
         }
     }
```

Result????

Naming standards for unit tests

- The name of the method should start with the word 'test'.
- The return type of the method should be null.
- The method shouldn't have any parameter.
- You should write a test case for the method which is less dependent on sub method. If you write a test case for dependent methods, your test may fail because the sub method can return wrong value. So, it may take longer to find out the source of the problem.
- Each unit test should be independent of other test. If you write one unit test for multiple methods then the result may be confusing.
- Test behavior is more important than methods. Each test case has to be in the same package as the code to be tested. For example if we want to create classes in root.baseclasses, then test cases must be in the same package.
- Although, it is nice to cover most of the code in the test cases, it is not advisable to use 100% of the code. If the method is too simple then there is no point of writing unit test for it.

Some Important Tips

- You should use object oriented best practice for your test code. You should bring all of the common functionalities in the base class. That way you can eliminate duplicate code, long methods etc.
- You should use proper method name. it is going to be easier for the team to maintain the code in future.
- The Junit framework automatically handles the exception. It is good idea not to throw an exception in a test that shouldn't throw an exception. Otherwise, it can hide bugs which may be discovered much later stage.
- JUnit doesn't provide all of the functionalities to test your code. So, you need an additional test tool. Some of the tools are DBunit, Selenium etc that can be used as an extension of junit.
- Instead of rely on System.out.println() method to figure out the success of the test, you can rely upon the 'assert' method of JUnit so that you can run the automated test and that way the tester doesn't need to monitor the test in person all the time.

The End

Thank You

? if any