Introduction to Software Engineering

Unit Testing with JUnit

Rami Sayar @ramisayar

GitHub Talks If you find a mistake, don't hesitate to fork the repository, clone it to your local machine, edit the mistake, `git commit -a -m` and then press the pull request button to notify me.

Introduction

The purpose of these notes is to introduce you to unit-testing with JUnit. JUnit is a test framework which uses annotations to construct tests and test suites. It is currently in its fourth version and is widely used in industry, practically the standard framework.

Some Principles of Testing

- Testing serves to ensure compliance with requirements. Failure is not meeting the requirements.
- Testing should be integrated from inception.
- Start with units and move to integration testing.
- The potential test search space is massive, only cover the most likely situations and edge cases.
- Tests are best written by some one who didn't write the code, who isn't biased.

Parameters for Testability

Good software is testable if the input and output can be observed and if they system can be decomposed to individual independent pieces.

Types of Testing

Unit-testing is when a test is conducted on a unit such as a class, routine, algorith, package or small program in the context of a much larger system. The unit-test is conducted in isolation from the rest of the system and tends to be written by the developers.

Integration testing is when a test is conducted on a group of units which interoperate. The group can be composed of two units or the entire system. Integration tests are also conducted by developers.

Regression testing is when you execute test cases previously passed and ensure that a change

did affect other parts of the software.

System testing is when you test the entire system to check if it meets functional requirements.

Acceptance testing is performed by the client to ensure the system meets their requiremetns and is ready to use.

Test Driven Development

TDD is an important part of agile software development processes and tends to follow these three rules. Source Wikipedia

1. Write tests first.
2. First fail the test cases.
3. Keep the unit small.

Unit-testing with JUnit

When unit-testing, you will typically have methods in a class which perform a specific test (test method) and typically that class will only contain test methods (test class)

Any class can serve as a test class, thus, JUnit will only look for classes which have the `@Test` annotation attached to a method.

```
@Test
public void testMultiply() {
   assertEquals("TEST CONDITION 50=50", 50, 50);
 }
```

JUnit assumes that all test methods are independent and can be executed in an arbitrary order.

JUnit Annotations

Annotations are used to tell JUnit what the responsibility of a method in a test class is.

- `@Test` executes a method as a test case in the test class.
- `@Before` executes a method once before each test in a class. `@After` executes a method once after each test in a class.
- `@BeforeClass` executes a method once before any test in a class. `@AfterClass` executes a method once after any test in a class.
- `@Ignore` ignores a test.
- `@Test(expected = Exception.class)` fails a test if it does not throw an exception.
- `@Test(timeout=100)` fails a test if it takes longer than 100 milliseconds.

The order of priority would look like this:

```
@BeforeClass - One Time Set Up
@Before - Set Up
@Test - Test
@After - Tear Down
@Before - Set Up
@Test - Test
@After - Tear Down
@AfterClass - One Time Tear Down
```

Test with Exception Example:

```
@Test(expected = ArithmeticException.class)
public void divideByZero() {
    int k = 1/0;
    fail();
}
```

JUnit Assert Statements

Assert statements are used to assert a test condition is true. They allow you to specify an error message, the expected result and the actual result.

- fail(string): Purposely fails a test with the string without checking a condition.
- assertTrue(message, condition): Checks if the condition is true.
- assertEquals(message, expected, actual): Checks if the expected and actual value are equal.
- assertEquals(message, expected, actual, tolerance): Checks if the expected and actual value are equal within a tolerance value. This is used to compare floating point values.
- assertNull(message, object) : Checks if an object is null. assertNotNull(message, object) does the opposite.

JUnit Test Suite

You can combine several test classes into one test suite and execute all the test cases at once.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ MyClassTest.class, MySecondClassTest.class })
public class AllTests {

}
```

Test Runner

To run a test class using Java instead of an IDE (such as eclipse), you can usie a test runner to execute the test class.

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class MyTestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(MyClassTest.class);
    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }
  }
}
```

References

http://www.vogella.com/articles/JUnit/article.html#unittesting
http://www.mkyong.com/tutorials/junit-tutorials/