Theory: JUnit and Mockito

22 minutes 0 / 5 problems solved

§1. Java Unit Testing

Java is an object-oriented programming language. One of the central concepts of Java is class. Any Java program can be seen as a combination of classes that interact with each other via methods. In Java, class is considered to be a unit, and methods are subject to unit testing.

Writing tests from scratch is a tedious task. Luckily, there are several unit testing frameworks that make this procedure easier. In this topic, we'll take a look at some of these frameworks.

§2. JUnit

JUnit is probably the most popular unit testing framework. It has a very active community that can answer your questions on forums. Let's consider the basic concepts of **JUnit**.

First, you need to include the **JUnit** library in your project. If you are using **maven** or **gradle** build systems, take the following templates and replace *VERSION* with the actual library version of your repository:

maven

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>VERSION</version>
    <scope>test</scope>
</dependency>
```

gradle

```
testCompile group: 'junit', name: 'junit', version: 'VERSION'
```

The method is interpreted as a test if it is tagged by *@Test* annotation:

```
@Test
public void testMethod() {
    ...
}
```

Usually, all unit tests of class methods are placed in a separate class. Build systems like **maven** or **gradle** include running unit tests at the end of the build process. It helps to detect bugs and fails the build if any of the unit tests fail.

One of the distinctive features of unit testing is the **isolation** of each test. To achieve that and avoid code duplication, you can use `@Before`, `@BeforeClass`, `@After`, and `@AfterClass` annotations.

Current topic:

JUnit and Mockito

Topic depends on:

- ✗ Unit testing
- ✓ Annotations
- ✗ Dependency management

Table of contents:

```
@BeforeClass
public static void setUpClass() throws Exception {
    // Code executed before the first test method
}

@Before
public void setUp() throws Exception {
    // Code executed before each test
}



@After

public void tearDown() throws Exception {

    // Code executed after each test

}



@AfterClass

public static void tearDownClass() throws Exception {

    // Code executed after the last test method

}
```

JUnit uses special API named **Assertions** as acceptance criteria. It is a set of methods that checks if the resulting value matches the expected value and throws an error if not. Some commonly used methods are:

- assertEquals
- assertTrue
- assertNotNull

Let's look at a simple example. Suppose we have a `Calculator` class which has two methods: `add` and `subtract`.

```
public class Calculator {

    public int add(int x, int y) {
        ...
    }

    public int subtract(int x, int y) {
        ...
    }

}
```

Here are the test examples with **JUnit:**

```
import org.junit.Assert;

public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 2);

        Assert.assertEquals(4, result);

    }



    @Test

    public void testSubtract() {

        Calculator calculator = new Calculator();

        int result = calculator.subtract(4, 2);



        Assert.assertEquals(0, result);

    }

}
```

In case of a failed assertion, the statement throws an `AssertionError` with a short description.

```
java.lang.AssertionError:
Expected :0
Actual   :2
```

§3. Mockito

In complex classes, a unit test often needs to instantiate and configure a lot of objects. The set-up is time-consuming and the test may turn out to be more complicated than the tested code itself. In these cases, **mockito** framework is particularly useful. It allows you to create a special object known as **mock** and define its behavior during execution.

To include **mockito** library into your project using **maven** or **gradle** build systems, use the following templates and replace *VERSION* with the actual library version of your repository:

maven

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>VERSION</version>
    <scope>test</scope>
</dependency>
```

gradle

```
testCompile group: 'org.mockito', name: 'mockito-core', version: 'VERSION'
```

Note that *mockito-all* is deprecated now.

Let's consider an example. `UsdConverter` class is responsible for converting local currency to USD (United States dollar). The test should check the behavior of the tested unit in isolation from its dependencies. As the exchange rate varies with time, `UsdConverter` uses `ExchangeRateService` to get the latest updates. Moreover, `getUsd` method of a live `ExchangeRateService` can send a request over **HTTP** to get the actual exchange rate, which is difficult to reproduce in the test environment. **Mockito** allows us to avoid those difficulties and provides an API for creating empty objects and managing behavior.

Note: here we use `BigDecimal` since financial transactions need high accuracy:

```
public class UsdConverter {

    private ExchangeRateService service;

    public UsdConverter(ExchangeRateService service) {
        this.service = service;
    }

    public BigDecimal convertToUsd(BigDecimal converted) {

        return converted.multiply(service.getUsd());

    }

}
```

Using **mockito**, we create a mock of `ExchangeRateService` object and define its behavior in case of `getUsd` method invocation:

```
import org.junit.Assert;
import org.mockito.Mock;

public class UsdConverterTest {

    @Mock

private ExchangeRateService service = Mockito.mock(ExchangeRateService.class);

    private UsdConverter converter = new UsdConverter(service);



    @Test

    public void testConvertToUsd() {

        Mockito.when(service.getUsd()).thenReturn(BigDecimal.valueOf(5));



        BigDecimal result = converter.convertToUsd(BigDecimal.valueOf(2));

        BigDecimal expected = BigDecimal.valueOf(10);



        Assert.assertEquals(expected, result);

    }

}
```

You can also throw an exception on a method call to test error handling. Another useful feature is checking that method was invoked inside the tested method.

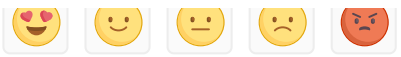
We have covered some of the most common **mockito** features; the full list of features is much wider than that.

§4. Conclusion

Usually, Java class is considered a unit and hence methods are subject to testing. Java community can offer a wide variety of tools to make unit testing easier; we have covered the two most common. **JUnit** framework allows you to prepare and clear the context of tests by annotations and provides API for checking acceptance criteria. **Mockito** framework helps with code isolation if you're testing complex classes and helps manage dependencies.

Report a typo

76 users liked this theory. **1** didn't like it. **What about you?**



Start practicing

Comments (2) Hints (0) Useful links (0) Show discussion