Xideral

Back End Academy

Mockito

Project By:

Kenneth Rodríguez García

Mockito

Summary

This project is a simple demonstration of the Mockito framework and how it can be used to control and verify the behavior of unit tests. In this case, we are making a Weight Loss Tracking and Prediction app, but we must first verify that unit tests are implemented properly. To achieve this, we will use Mockito to simulate the behavior of our “Weight Loss Predicition Algorithm” and verify that it produces the expected results.

Research

Unit Testing

Unit testing is the practice of testing individual methods to ensure each one works as expected. These tests are designed to verify that each method performs its task correctly and independently of the rest of the system.

By doing this practice, developers can identify bugs earlier and easier, which also helps to improve the quality of the code base.

JUnit

JUnit is a framework used to write automated unit tests in Java, allowing an easier testing process. The framework uses annotations, which can be identified by the ‘@’ character, to mark the test methods. When running a unit testing task, all methods with the “@Test” annotation will be tested, without needing to call them explicitly.

To verify that the methods produce the expected behavior, JUnit provides assertion methods, which compare the result with the expected value.

Mockito

Mockito is a Java framework used to create mock objects for unit testing. In essence, the Mock objects are simulated objects which can be controlled and verified easily.

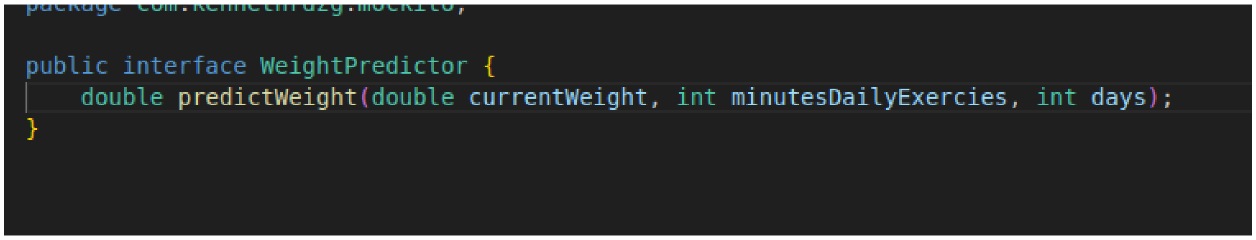
This framework is a valuable tool for unit testing, as it provides a way of making tests more isolated and maintainable.

Design

Our program consists of very simple classes to better demonstrate how Mockito is useful for unit testing.

Interface - WeightLossPredictor

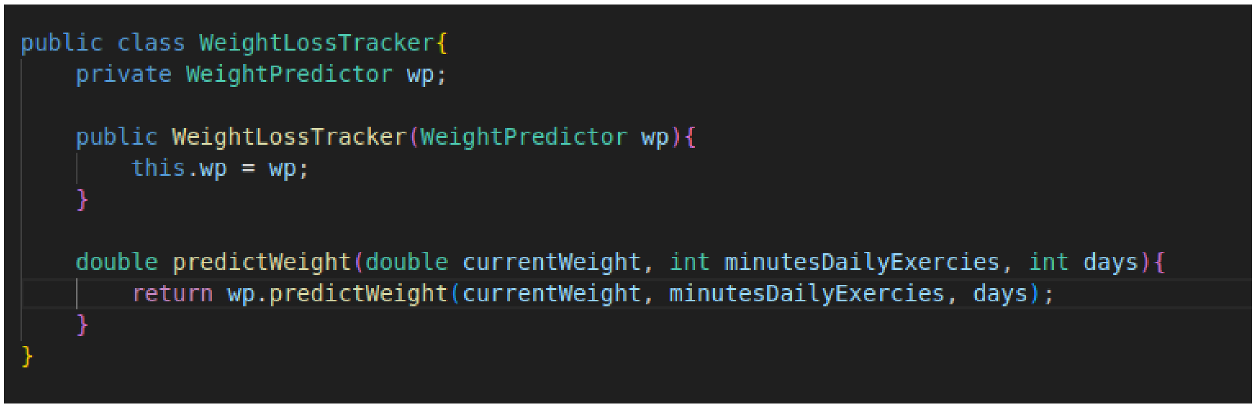
This interface is what the Mock objects will use to simulate behavior. Instead of actually implementing a method, we only need to define it.



```java
package com.kennethrdzg.mockito;

public interface WeightPredictor {
    double predictWeight(double currentWeight, int minutesDailyExercies, int days);
}
```

Class - WeightLossTracker

This class will implement the prediction method, so it must have a WeightPredictor attribute.



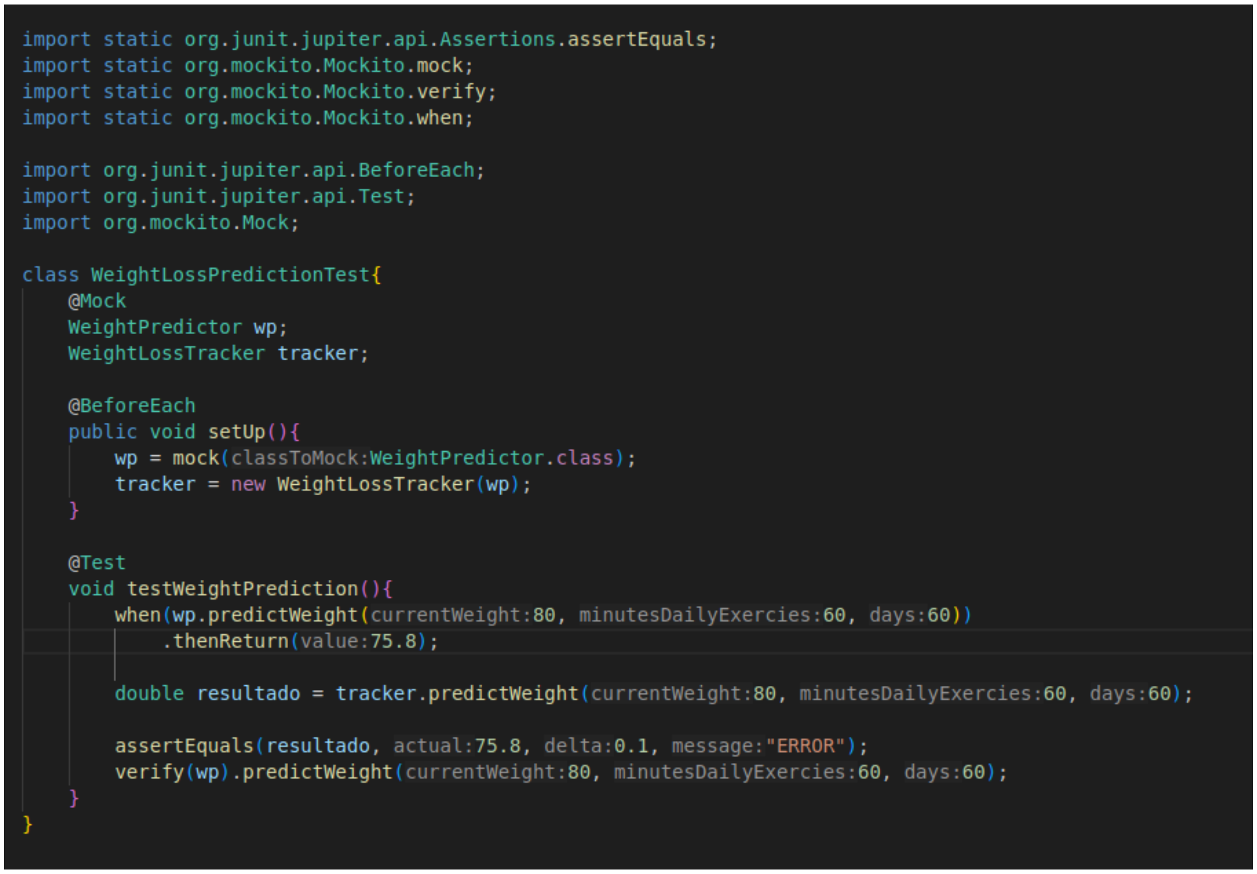
```java
public class WeightLossTracker{
    private WeightPredictor wp;

    public WeightLossTracker(WeightPredictor wp){
        this.wp = wp;
    }

    double predictWeight(double currentWeight, int minutesDailyExercies, int days){
        return wp.predictWeight(currentWeight, minutesDailyExercies, days);
    }
}
```

Test - WeightLossPredictionTest

This class contains the unit tests to perform, as well as the Mock objects to use.

The “@Mock” annotation signals thath the WeightPredictor object will be a Mock object, while the “@BeforeEach” annotation executes the “setUp()” method before each unit test.

Finally, the “@Test” annotation is used to mark each of the different Unit Tests (only one in our case).

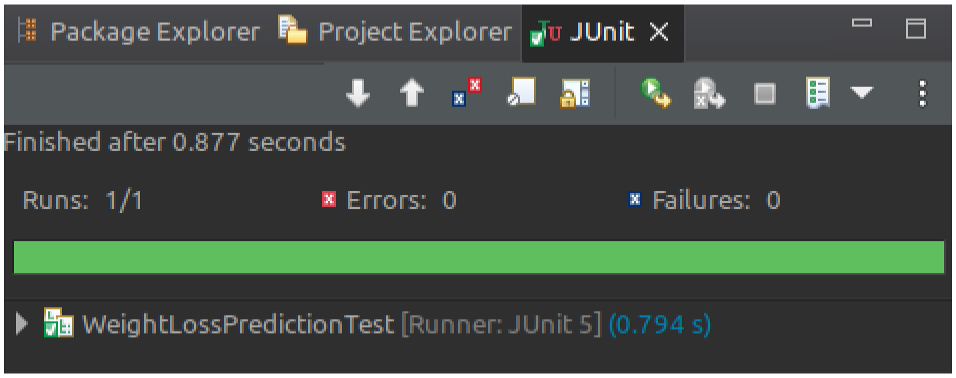


```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;

class WeightLossPredictionTest{
    @Mock
    WeightPredictor wp;
    WeightLossTracker tracker;

    @BeforeEach
    public void setUp(){
        wp = mock(classToMock:WeightPredictor.class);
        tracker = new WeightLossTracker(wp);
    }

    @Test
    void testWeightPrediction(){
        when(wp.predictWeight(currentWeight:80, minutesDailyExercies:60, days:60))
            .thenReturn(value:75.8);

        double resultado = tracker.predictWeight(currentWeight:80, minutesDailyExercies:60, days:60);

        assertEquals(resultado, actual:75.8, delta:0.1, message:"ERROR");
        verify(wp).predictWeight(currentWeight:80, minutesDailyExercies:60, days:60);
    }
}
```

Execution

By executing the WeightLossPredictionTest class, our IDE will execute all defined unit tests.

In our case, the unit test will create a Mock WeightPredictor object and set its expected behavior, allowing for a much simpler comparison and verifying that the Unit test has been set up properly.



```
Finished after 0.877 seconds

Runs:  1/1          Errors:  0          Failures:  0

WeightLossPredictionTest [Runner: JUnit 5] (0.794 s)
```

As we can observe, the unit test as executed succesfully, resulting in 0 error and 0 failures.

Conclusion

I am still relatively new to the practice of unit testing, but Mockito seems to be an interesting tool to use. I've spent my fair share of time setting up objects with specific attributes to verify that their behavior was what I expected, but the task is always very time consuming. Mockito appears a great solution to this, allowing developers to write proper unit tests and improve their efficiency.