SOFT3202

Adam Ghanem

March 9, 2022

Contents

1 Tutorial 2 EXPORT

1.1 JUnit Testing

1.1.1 Revision

Let us breakdown a live method and see from where we can test and what is wrong with it:

```
boolean checkCollision(double boundsX, double boundsY,
                       double circleOneX, double circleOneY, double circleOneRadius,
                       Double circleTwoX, Double circleTwoY, Double circleTwoRadius)
```

What can we see from this:

1.1.2 Answer

- null Doubles
- x coordinates out of bounds
- y coordinates out of bounds
- negative radii
  - check valid collisions
  - check no collisions
  - check inside one other
  - check same circle
  - check one in bounds other isn't
  - many more

This method could be tested using a whole test suite. Realistically you would want to abstract the method above to make it easier to work with and generally more maintable.

1.1.3 Timeouts

Later on we will learn this, but I would like to go through timeouts with JUnit. Using this we can actually set a timeout like the ones you experience on Edstem.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class ShoppingBasketTest {

  // The test will fail if it exceeds the limit
  // These are all measured in milliseconds
  @Test (timeout = 100)
  void testGetItemsValid() {

    // Notice that no asserts are really necessary
    ShoppingBasket sb = new ShoppingBasket();
    sb.addItem("orange", 10);
    sb.getItems();
  }

  @Test(timeout=100)
  void failTimeout() {
    ShoppingBasketTest sbt = new ShoppingBasketTest();
    sbt.wait(100);
  }
}
```

1.2 Verification vs Validation

As I mentioned I would just like to quickly go over this quickly:

- Validation is the idea that we must **build the program according to the specification**.
  - it makes us the question of *Have we fulfilled that the customer's needs are met by our specification*
- Verification instead ensures of have we built the right thing given the specification
  - it is making sure that our software coalesces with the specification.

1.3 Agile and Waterfall

Simple differences here (Josh goes over them in more detail): Agile:

- daily testing
- standup meetingsa
- constant feedback loops

Waterfall:

- separated into stages
- in that stage we perform tasks specific to it e.g. requirements stage will gather requirements, verification stage will test the system and etc.

1.4 Mockito

Mockito is the fancy new library that works with JUnit. As we know we have multiple types of test doubles from:

These three are most similar to each other:

- stubs
  - just return defaults
- dummy
  - pass object that is empty or unused
- fake
  - like stub but the object is used and has a simpler implementation

These two are most similar to each other:

- mocks
  - we will explain below...
- spy
  - captures and monitors indirect output calls to verify correctness

But why do we need mocks, let us take a look at a code example from Canvas:

```
public class MyClass {
  private MyDependency dependency;

  public MyClass(MyDependency dependency) {
    this.dependency = dependency;
  }

  public int getSomething() {
    return dependency.getSomething();
  }

  public void inputSomething(int value) {
    dependency.inputSomething(value);
  }
```

```
}

@Test
public void testGetSomething() {
  // Mock the dependency
  MyDependency md = mock(MyDependency.class);
  // Provide the behaviour
  when(md.getSomething()).thenReturn(120);
  // Create the class and inject dependency
  MyClass mc = new MyClass(md);
  // Assert that statement
  assertThat(mc.getSomething(), 120);
}

public class MyDependency {
  private int baseNumber = 0;

  public int getSomething() {
    return baseNumber;
  }

  public void inputSomething(int value) {
    this.baseNumber = value * 10;
  }
}

public class MyDependencyMock extends MyDedependency {
  private int number;
  public int getSomething() {
  }
}

public class StorageMock implements Storage {
    private String lastCalledAddElementValue = null;
    private String[] returnValueForGetElements = new String[] {};
    @Override
    public void addElement(String element) {
        lastCalledAddElementValue = element;
    }
    @Override
    public String[] getElements() {
        return returnValueForGetElements;
    }

    public String getLastCalledAddElementValue() {
        return lastCalledAddElementValue;
    }

    public void setReturnValueForGetElements(String[] value) {
        this.returnValueForGetElements = value;
    }
}


public class StorageConsumerTest {
  @Test
  public void testAddElementCalledRightParameters() {
    StorageMock sm = new StorageMock();
    StorageConsumer sc = new StorageConsumer(sm);
    sc.addElement("oh hi mark");

    assertThat(sm.getLastCalledAddElementValue(), is(equalTo("oh hi mark")));
  }
 }
```

A mock allows us to test objects whose methods depend on something else or another object. I know there were questions raised regarding what to do if a method like `getItems()` requires the use of another method to add items. It allows us to stay true to the Single Responsibility Principle.

Key point: we use mockito to create a mock or known implementation of the dependency to **test our object in isolation.**

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.equalTo;
```

```
public class ShoppingBasketTest {


  // Assuming this method exists in ShoppingBasket
  public static double calculateCost(ShoppingBasket sb) {
  }

  // Calculating the total cost of the basket
  @Test
  void calculateCostValid() {
    // Create the mock of the dependency
    ShoppingBasket sb = mock(ShoppingBasket.class);

    // Create a list to mimic the items list
    ArrayList<Pair<String, Integer>> items = new ArrayList<>();
    items.add(new Pair<String, Integer>("apple", 619));

    // When the shopping basket getItems is called it will return this list
    when(sb.getItems()).thenReturn(items);

    // Assert that it has calculated the correct cost
    assertThat(ShoppingBasket.calculateCost(sb), is(equalTo(1238)));

    // Verify that the method has been called once
    verify(sb, times(1)).getItems();
  }
}
```

But is that all mockito can do?

- mockito will/can create a spy of our object as well
- plus it can do a 1000 other things to help mimic our objects

Under the hood Mockito will look to extend the dependency and override the methods that we define a `thenReturn` with. If it uses `extends` then it prohibits us to use Mocks with what?

We divide Mocking or the application of it into three main questions:

- what do we need to Mock and when should we Mock?

- what to verify?
- what can't we mock

1.4.1 What to Mock and when?

As mentioned previously, mocking is useful when we have tests **that have dependencies on other sections of our codebase** which inhibits our ability to test them in isolation. We should mock the dependency but only if we have direct control/ own it (will explain this later).

As an exercise we should try and mock a Stock class and get it to work!

```
// We can do a stub implementation or we could mock the entire thing
public interface Stock {
  public double getPrice();
}

  public class Trader {

    public double buyStock(Stock stock, int amount) {
      double price = stock.getPrice();
      return price * amount;
    }
  }

  public class TraderTest {

    @Test
    public void testBuyStockValidOne() {
      Stock stock = mock(Stock.class);

      // Set the verified output for this mock when the getPrice method is called
      when(stock.getPrice()).thenReturn(120.00);

      // Set initial balance
      Trader trader = new Trader(1000);
      assertEquals(120, trader.buyStock(stock, 1) );
    }
  }
```

On top of using `thenReturn` we can make use of `thenThrow` to mock exceptions being thrown.

```
@Test
public void testBuyStockOutOfStockQuery() {
  // We can also specify throwing exceptions into our behaviour
  when(stock.getPrice()).thenThrow(new OutOfStockException());

  OutOfStockException thrown =
    Assertions.assertThrows(
                           OutOfStockException.class,
                           () -> {
                             traderInjected.buyStock(stock, 0);
                           },
                           "OutOfStockException was expected");

}
```

Instead of having those specific `mock` methods we can use an **annotation, `@Mock`.** And wouldn't it be great to also be able to insert and inject the mocks into the class that uses the dependency?

Well look no further, you have been gifted an early (or belated, or on-time!) birthday present. With `@InjectMocks` we can insert our previously created `@Mock` annotations into *constructor, setter or other fields.*

```
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class TraderTest {

  // We can make use of annotations to mock
  @Mock
  private Stock stock;

  @InjectMocks
  private Trader trader;

  // The other way is using @BeforeEach
  private Stock beforeStock;
  private Trader beforeTrader;

  @BeforeEach
  public void setUp() {
    stock = mock(Stock.class);
    trader = new Trader();
  }

  @Test
  public void testBuyStockBuyingTheDip() {
    // No need to have the above settings
    when(stock.getPrice()).thenReturn(10);
    assertEquals(10000, trader.buyStock(stock, 1000));
  }
}
```

1.4.2 What to verify?

Verify is a bit different to the assertions we are used to. Verify will make sure that the code was exectured correctly in agreement with the specifications. For example we can test whether the return result was valid but *we verify if the right methods were called during this test.*

```
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class TraderTest {

  @Mock
  private Stock stock;

  @InjectMocks
  private Trader traderInjected;

  @Test
  public void testBuyStockTwoStocks() {
    // Set the verified output for this mock when the getPrice method is called
    when(stock.getPrice()).thenReturn(100.00);
    // Set initial balance
    Trader trader = new Trader(1000);

    // We can verify if they method for stock was called one time.
    verify(stock, times(1)).getPrice();
    assertEquals(200, trader.buyStock(stock, 1) );
  }


  @Test(expected=IllegalArgumentException.class)
  public void testBuyStockZeroStocks() {
    when(stock.getPrice()).thenReturn(120.00);
    traderInjected.buyStock(stock, 0);

    // Ensure the stock was never queried
    verify(stock, never()).getPrice();
  }


}
```

If we don't care about which arguments were passed through when verifying we can use:

- `anyInt()`, `anyString()` or `any(<CLASS>.class)`
  - `class.parseName(anyString())`
- these are known as ArgumentMatchers, and there is a lot more you can do with them.

Among this we can verify if:

- a method was called 0, 1, or n number of times
- whether it was called with the right arguments
- verify there are no more interactions through `verifyNoMoreInteractions()`, used after all the `verify` calls.
- verifying the order of calls with `inOrder()`

1.4.3 What we shouldn't Mock?

The main message here is more of a warning, and it is also provided on the Mockito github documentation.

- IT IS NOT TO MOCK A TYPE WE DO NOT OWN

This means that if we are trying to mock an external API or class our test suite will not be aware of future changes they may make. If we have mocked this third party and pass our tests that is not indicative that we would pass when the software is deployed. Moreover when trying to mock this we may find that we have to fill in a plethora of different mocks just to get a simple test working. In general, as a rule of thumb, *if we do not have access to the API then we should avoid mocking it.*

Instead they suggest to make use of wrappers of the external API, that we have control of, in order to test interaction with it and mock it.

I recommend this reading, it is short, and explains making clear and effective tests. The page also offers relevant links to other developers who have blogged on the issue.

1.4.4 Spy

What is a Spy?

- a spy is another form of test double that can mask an existing method of an object while still being able to track its interactions.

And again we can mke use of annotations to make it much clearer.

```
@ExtendWith(MockitoExtension.class)
public class ShoppingBasketTest {
  public void spyOnShoppingBasket() {
    ShoppingBasket spyBasket = spy(new ShoppingBasket());

    spyBasket.addItems("apple", 10);

    // Verifying that the addItems method was called once with those parameters
    // Make sure to specify the number of times
    verify(spyBasket, times(1)).addItems("apple", 10);

    // Asserting that we have 10 apples
    // Assuming here getItem will just return the number of apples
    assertEquals(10, spyBasket.getItem("apple"));
  }

  @Spy
  private ShoppingBasket spyBasketAnnotated;

  public void spyOnShoppingBasket() {
    spyBasketAnnotated.addItems("apple", 10);

    // Verifying that the addItems method was called once with those parameters
    // Make sure to specify the number of times
    verify(spyBasketAnnotated, times(1)).addItems("apple", 10);

    // Asserting that we have 10 apples
    // Assuming here getItem will just return the number of apples
    assertEquals(10, spyBasketAnnotated.getItem("apple"));
  }
}
```

Mockito will create our mocks and spies when calling a mock method. We can also create methods for our spy based on certain scenarios.

1.4.5 Inorder Example

```
@Test
public void testInOrderExample() {
  // We can also specify throwing exceptions into our behaviour
  TheBatman batman = mock(TheBatman.class);
  hollywood.makeMovie(batman);
  InOrder inOrder = inOrder(batman);
  inOrder.verify(batman, times(1)).foo("Batman Begins");
  inOrder.verify(batman, times(1)).bar("Batman Dark Night");
  inOrder.verify(batman, times(2)).bar("Dark Night Rises");
}
```