Group 33
Víctor Rosa
Martín Ordoño

Table of Contents

1. Unit Test

A unit test should test functionality in isolation. Side effects from other classes or the system should be eliminated for a unit test, if possible.

This can be done via using test replacements (*test doubles*) for the real dependencies. Test doubles can be classified like the following:

- A ***dummy object*** is passed around but never used, i.e., its methods are never called. Such an object can for example be used to fill the parameter list of a method.

- ***Fake*** **objects** have working implementations, but are usually simplified. For example, they use an in memory database and not a real database.

- A ***stub*** **class** is an partial implementation for an interface or class with the purpose of using an instance of this stub class during testing. Stubs usually don't respond to anything outside what's programmed in for the test. Stubs may also record information about calls.

- A ***mock object*** is a dummy implementation for an interface or a class in which you define the output of certain method calls. Mock objects are configured to perform a certain behavior during a test. They typically record the interaction with the system and tests can validate that.

  Test doubles can be passed to other objects which are tested. Your tests can validate that the class reacts correctly during tests. For example, you can validate if certain methods on the mock object were called. This helps to ensure that you only test the class while running tests and that your tests are not affected by any side effects.

1.1 Mock Object Generation

You can create mock objects manually (via code) or use a mock framework to simulate these classes. Mock frameworks allow you to create mock objects at runtime and define their behavior.

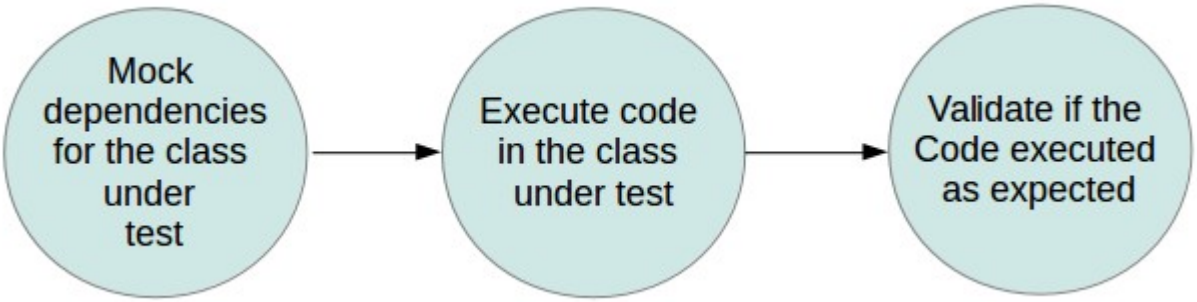
The classical example for a mock object is a data provider. In production an implementation to connect to the real data source is used. But for testing a mock object simulates the data source and ensures that the test conditions are always the same.
These mock objects can be provided to the class which is tested. Therefore, the class to be tested should avoid any hard dependency on external data.
Mocking or mock frameworks allows testing the expected interaction with the mock object. You can, for example, validate that only certain methods have been called on the mock object.

1.2 Using Mockito for mocking objects

Mockito is a popular mock framework which can be used in conjunction with JUnit. Mockito allows you to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly.

If you use Mockito in tests you typically:

- Mock away external dependencies and insert the mocks into the code under test
- Execute the code under test
- Validate that the code executed correctly



In our project we are goint to use mockito in the class **BrotherhoodControllerTest.java**

2. Configurations

To use Mockito we must to follow the next steps:

1. Adding maven dependencies directly in the pom.xml file

```
<!-- Mockito -->

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-core</artifactId>
    <version>1.3</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-library</artifactId>
    <version>1.3</version>
    <scope>test</scope>
</dependency>
```

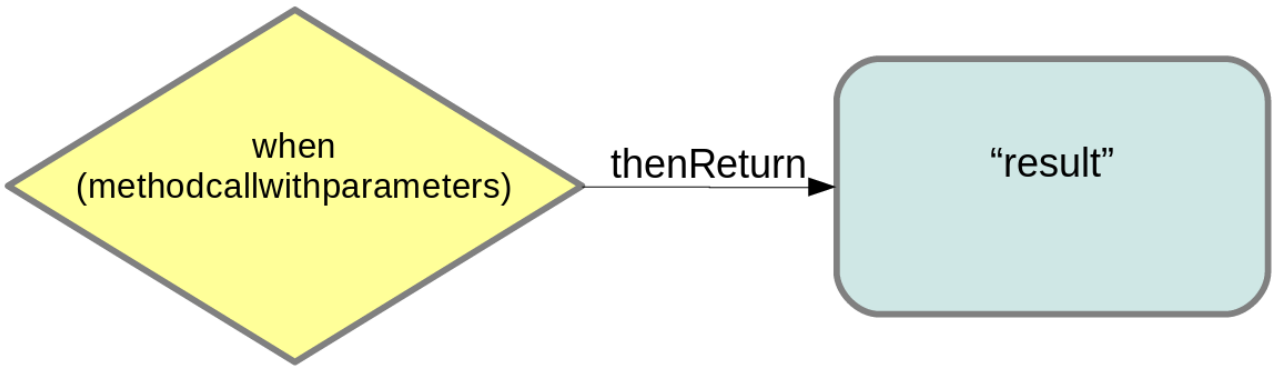
2. Imports in our test class.

```
import static org.mockito.Mockito.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
import static org.hamcrest.Matchers.*;
```

3. Mockito Annotations

- Using the **@Mock** annotation to create the mock object.
- Using the **@InjectMocks** annotation for dependency injection via Mockito.
- Using the "**when.. thenReturn**" statement.

Mocks can return different values depending on arguments passed into a method. The `when(…).thenReturn(… )` method chain is used to specify a a return value for a method call with pre-defined parameters.



4. BrotherhoodControllerTest.java

The following class demostrate the usege of the different annotations.

BrotherhoodControllerTest.java

```java
@RunWith(MockitoJUnitRunner.class)
public class BrotherhoodControllerTest {

    @Mock
    private BrotherhoodService brotherhoodService;

    @Mock
    private ConfigurationsService configurationsService;

    @InjectMocks
    private BrotherhoodController brotherhoodController;

    private MockMvc mockMvc;

    @Before
    public void setup(){
        // Process mock annotations
        MockitoAnnotations.initMocks(this);

        // Setup Spring test in standalone mode
        this.mockMvc = MockMvcBuilders.standaloneSetup(this.brotherhoodController).build();

    }
    /**
     *
     * Prueba que la lista de Brotherhoods que crea el servicio y utiliza el controllador.
     */

    @Test
    public void testListMessages() throws Exception{
        ArrayList<Brotherhood> brotherhoods = new ArrayList<>();
        brotherhoods.add(this.brotherhoodService.create());
        brotherhoods.add(this.brotherhoodService.create());

        String title = "title";
        String logo = "https://tinyurl.com/acme-madruga";

        Configurations config = new Configurations();
        config.setTitle(title);
        config.setLogo(logo);

        // Config from AbstractController
        when(this.configurationsService.getConfiguration()).thenReturn(config);

        // Check BrotherhoodController
        when(this.brotherhoodService.findAll()).thenReturn(brotherhoods);

        this.mockMvc.perform(get("/brotherhood/list"))
                    .andExpect(status().isOk())
                    .andExpect(view().name("brotherhood/list"))
                    .andExpect(model().attribute("brotherhoods", hasSize(2)));

    }
}
```