Mockito: Basics

Development:

Marcelo Eduardo Guillen Castillo

Java Monterrey Academy

September 5,, 2024

Introduction

Business case

Java projects need to be tested, it is very important to verify that our codes are fine and functional, but sometimes if we involve classes and more complex objects we are going to have conflicts, that's why Mockito exists, a dependency for Java projects where you can create more sophisticated tests like instance injection and simulated data.
Our team (named team F) is in charge of the new project for our product management, an API that could make use of our Product Management System for Java language, the problem is that the development is still making its first steps and the CEO of the company already wants like a “mock” of the idea of the API. So our company needs some tests in order to make sure our code at the moment is still making sense.

Objectives

Demonstrate to the CEO how our API should work and its main implementation, even if it even hasn’t implemented a real class that could create an instance.
Help us analyze as developers the way to implement new functionalities and applying design patterns for our tool.

Requirements

- Every time we create an instance of the class ActiveOnUse, we need an instance of the interface ActiveCloudOperation.
- Make a calculation of the value of the active on use depending on its value market on the cloud.
- Obtain the state of the active on the cloud at any moment.

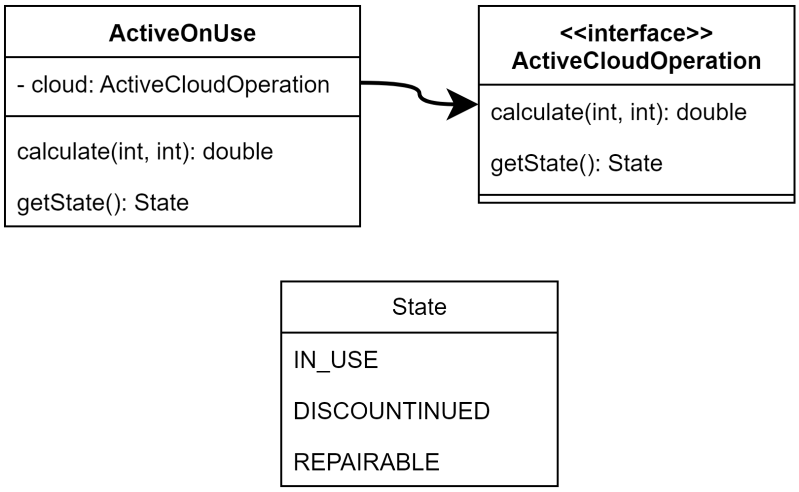
Installation

The engineer needs to install the JDK 17 to beyond, then any IDE that could maintain Maven projects, and finally install all the dependencies and plugins necessary to make the project work correctly. This is made in Maven, so please enter the following code into your POM file.

```
Unset
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
       <modelVersion>4.0.0</modelVersion>
       <groupId>group</groupId>
       <artifactId>MockitoTry</artifactId>
       <version>0.0.1-SNAPSHOT</version>

       <dependencies>
              <dependency>
                     <groupId>org.junit.jupiter</groupId>
                     <artifactId>junit-jupiter-api</artifactId>
                     <version>5.8.0</version>
                     <scope>test</scope>
              </dependency>
              <dependency>
                     <groupId>org.mockito</groupId>
                     <artifactId>mockito-core</artifactId>
                     <version>5.13.0</version>
              </dependency>
              <dependency>
                     <groupId>org.projectlombok</groupId>
                     <artifactId>lombok</artifactId>
                     <version>1.18.34</version>
                     <scope>provided</scope>
              </dependency>
       </dependencies>
</project>
```

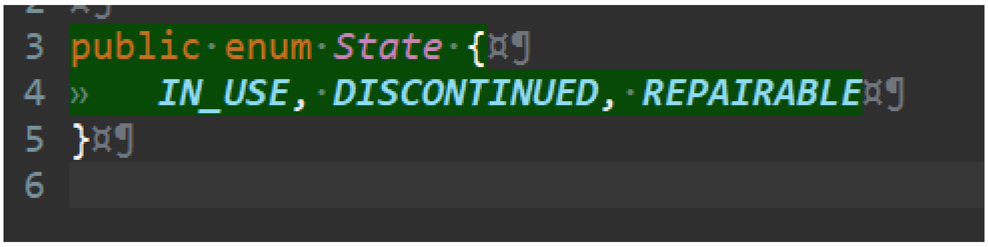
Project architecture

An active can be in use on that moment, so we assign it to an instance of ActiveOnUse, which can have access to ActiveCloudOperation interface, which has the intention to be a bunch of family functions that could grant access to some cloud services related to actives, but with the actual state of the team, it's still on development.

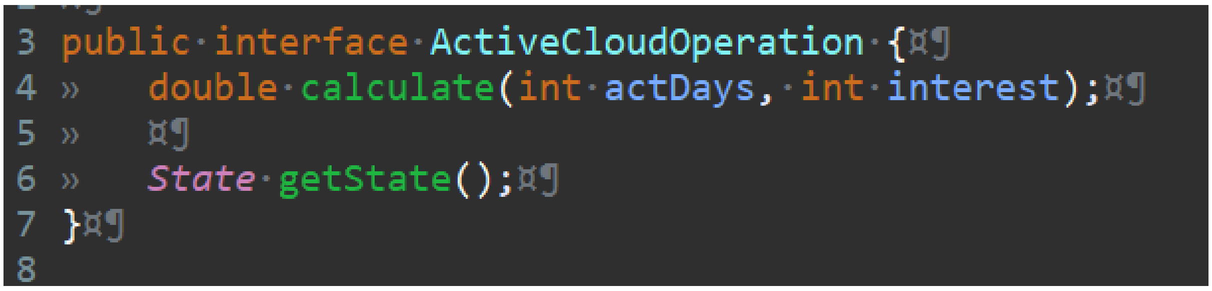


Demonstrations

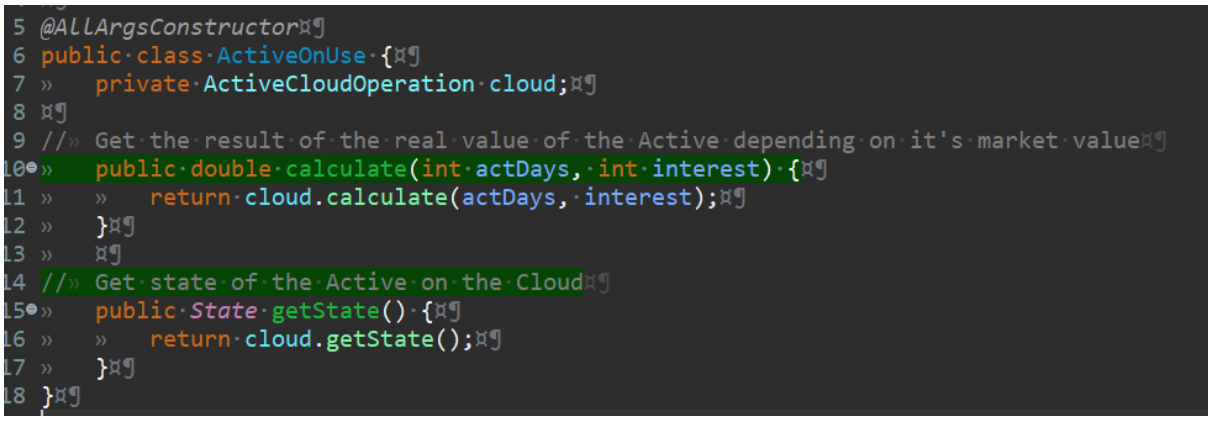
Here are some demonstrations using mockito's functionalities.
First of all we need to create an enumeration of which states exists on a product or "active" the name we use on these projects.


```java
public enum State {
	IN_USE, DISCONTINUED, REPAIRABLE
}
```

Create an interface that indicates all the functionalities that will access into the cloud of our inventory that will grant access to any product that is in use.

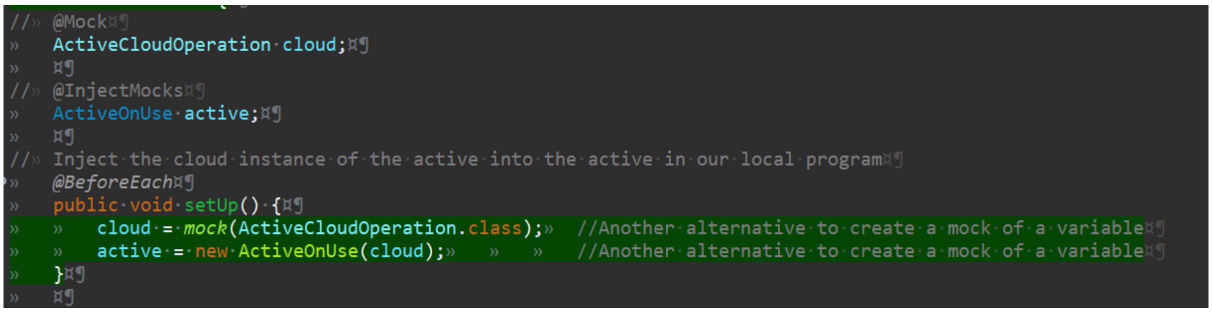

```java
public interface ActiveCloudOperation {
	double calculate(int actDays, int interest);
	
	State getState();
}
```

Then we need to create a class that represents the product on use for the local company, which will soon need an injected ActiveCloudOperation instance named "cloud", and also implementing it's methods.



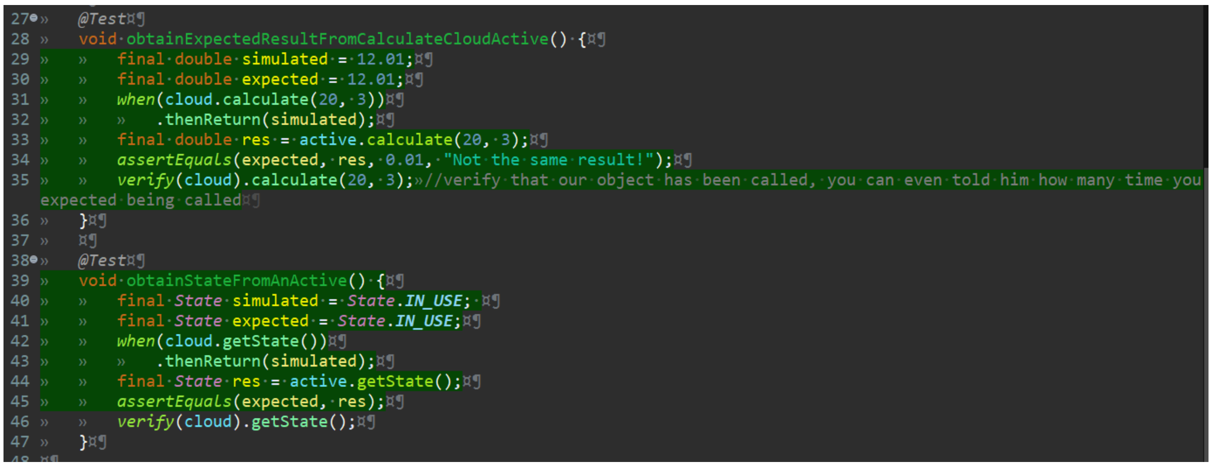
```java
@AllArgsConstructor
public class ActiveOnUse {
	private ActiveCloudOperation cloud;

//	Get the result of the real value of the Active depending on it's market value
	public double calculate(int actDays, int interest) {
		return cloud.calculate(actDays, interest);
	}
	
//	Get state of the Active on the Cloud
	public State getState() {
		return cloud.getState();
	}
}
```

Then we need to create the mockito tests, we need to inject the cloud interface an instance and then inject it into the ActiveOnUse object.



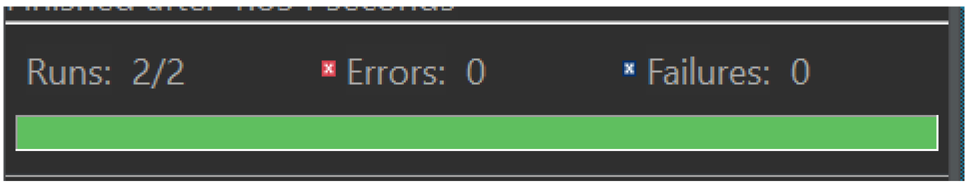
```java
//	@Mock
	ActiveCloudOperation cloud;
	
//	@InjectMocks
	ActiveOnUse active;
	
//	Inject the cloud instance of the active into the active in our local program
	@BeforeEach
	public void setUp() {
		cloud = mock(ActiveCloudOperation.class);	//Another alternative to create a mock of a variable
		active = new ActiveOnUse(cloud);		//Another alternative to create a mock of a variable
	}
	
```

Finally we make the tests that will simulate an implementation of our interface.

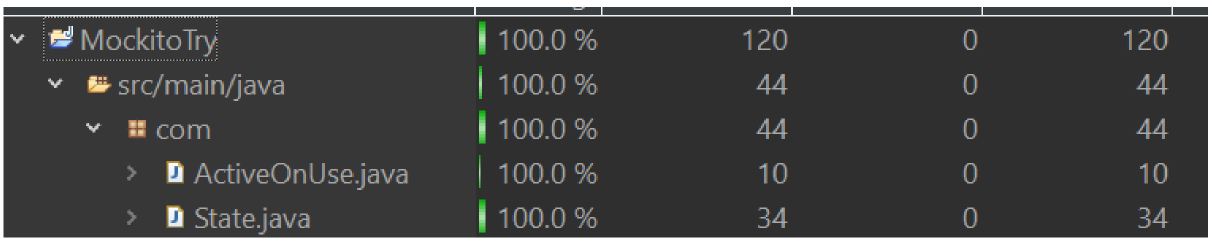


```java
	@Test
	void obtainExpectedResultFromCalculateCloudActive() {
		final double simulated = 12.01;
		final double expected = 12.01;
		when(cloud.calculate(20, 3))
			.thenReturn(simulated);
		final double res = active.calculate(20, 3);
		assertEquals(expected, res, 0.01, "Not the same result!");
		verify(cloud).calculate(20, 3);//verify that our object has been called, you can even told him how many time you expected being called
	}
	
	@Test
	void obtainStateFromAnActive() {
		final State simulated = State.IN_USE; 
		final State expected = State.IN_USE;
		when(cloud.getState())
			.thenReturn(simulated);
		final State res = active.getState();
		assertEquals(expected, res);
		verify(cloud).getState();
	}
```

Then we can see the results.



```
Runs:  2/2          Errors:  0          Failures:  0
```



MockitoTry	100.0 %	120	0	120
src/main/java	100.0 %	44	0	44
com	100.0 %	44	0	44
ActiveOnUse.java	100.0 %	10	0	10
State.java	100.0 %	34	0	34

Conclusion

A software project will always need a demonstration of its possibilities, specially on the DevSecOps cycle of verification of functionalities and methods. It is also important to document unit tests on our projects, make an analysis of the possible vulnerabilities and errors our code could have, and most importantly what coverage our test takes for our code, and finally release a finished product.