UnitTesting - Mocking - Code Coverage

Unit testing is a software testing method where individual components or functions of a program are tested in isolation to ensure they work as intended. This process involves writing test cases for each small part of the application, which helps identify and fix bugs early in the development cycle. Performing unit testing is crucial because it improves code quality, simplifies debugging, and facilitates future maintenance. By catching errors at an early stage, developers can ensure that changes in the code do not introduce new issues, leading to a more stable and reliable software product.

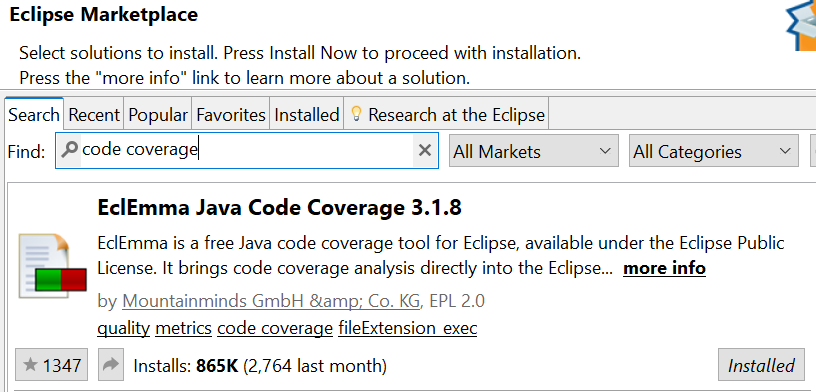
Reference:

<https://chatgpt.com/share/e67ae3bd-754f-438e-a090-f62983ee77ac>

**Key Bullets:**

1. Every possibility of code should be testing.
2. Every class should be tested independently.
3. Mock any outside dependent class. If you create instance, you cannot mock it. It has to be an instance variable.
4. In code coverage, try to be 80% during testing.
5. You can test every code in the local machine.
6. For many projects without proper code coverage, your code merge will not be accepted.
7. When you are unit-testing you are not actually using SpringBoot Container. We only run the test-class & nothing else.
8. There is no mechanism for component scanning, there is no spring container, which means there are no beans to inject.

**Steps to apply unit testing:**

1. We are using Junit & Mockito for Unit testing & Mocking
   1. 
2. Add Junit & Mockito dependency in pom.xml. (For starting up project, it's already added)
3. Creating TestClass for Testing
   1. For every functional class (with business logic) create test class in src/main/test.
   2. Follow same package structure for functional class & test class.
   3. Name of test class, <FUNCTIONAL CLASS NAME>Test.java
   4. If functional class PaymentServiceImpl.java, then test class is PaymentServiceImplTest.java
4. Annotate test class with @ExtendWith(MockitoExtension.class)
   1. @ExtendWith(MockitoExtension.class)
   2. public class PaymentServiceImplTest
5. Create instance of functional class in test class. And annotate with @InjectMocks
   1. @InjectMocks
   2. private PaymentServiceImpl paymentService;
6. Mock any external class that is used in PaymentServiceImpl using @Mock
   1. @Mock
   2. private ApplicationContext context;
7. Add a test method for testing. Annotate the method with @Test. @Test tells JUnit to run the method as a test.

## **How to write a unit test [The method annotated with @Test]:**

* 1. Arrange: Mock the data needed to carry out the unit test.
  2. Act: Call the method that you want to test.
  3. Assert: Check if the method is behaving correctly.

## **Arrange: Mock the data needed to carry out the unit test.**

* Create proper Java objects necessary for the test-case to run.
* Mock the external classes & method, for test-case to run.

### **Using Reflection to set values to instance fields**

* ReflectionTestUtils.setField(paymentService, "validators", "CHECK\_1,CHECK\_2");

### **Setting expectations to Mock**

* when(appContext.getBean(any(Class.class))).thenReturn(validationBean);
* when(paymentStatusService.processStatus(any(TransactionDTO.class))).thenReturn(txnDTO);
* when(transactionDao.getTransactionByReference(anyString())).thenReturn(null);
* when(userService.createUser(any())).thenReturn(123l);
* when(httpRestTemplateEngine.execute(any())).thenReturn(transactionResponse, paymentResponseEntity);
* *when*(**this**.gson.fromJson(*anyString*(), *any*(Class.**class**))).thenReturn(orderDTO);
* *when*(modelMapper.map(*any*(CreatePaymentRes.**class**), *eq*(InitiatePaymentResDTO.**class**))).thenReturn(resDto);
* doNothing().when(validator).doValidate(any());
* Throw Exception
  + doThrow(new ValidationException("Validation failed")).when(validationBean).doValidate(paymentRequest);
  + when(signatureCreatorMock.generateSignature(plainText)).thenThrow(YourExpectedException.class);
* Static Mocking
  + MockedStatic<ValidatorEnum> mockedStatic = Mockito.mockStatic(ValidatorEnum.class);
  + mockedStatic.when(() -> ValidatorEnum.getEnumByName(Mockito.anyString())).thenReturn(ValidatorEnum.PROVIDER\_ID\_FILTER);

## **Act: Call the method that you want to test.**

## 

## 

## **Assert: Check if the method is behaving correctly.**

### **Expecting Exception from method:**

ProcessingException processingExp = assertThrows(

ProcessingException.class, () ->

paymentServiceImpl.initiatePayment(txnRef, reqDto));

### **Expecting No Exception**

assertDoesNotThrow(() -> {

return duplicateTransactionValidator.doValidate(req);

});

### **Other commonly used Assertions**

* assertEquals
* assertNotEquals
* assertFalse
* assertTrue
* assertNull
* assertNotNull

### **Count occurrence of method**

verify(mock, times(1)).method()

The Mock

Number of times you expect its method to be invoked

The method