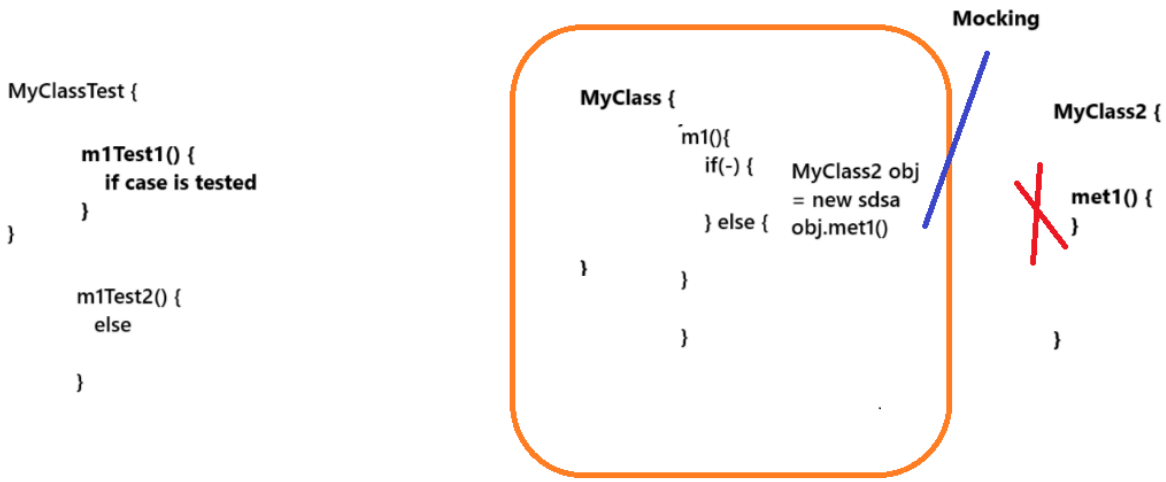
DIAGRAM:



Unit Testing:

1st we complete functional development.
UnitTesting
(you write additional code to test your functional code)
Dev env testing.

==========

TDD
functional dev is driven by writing testcases
Dev env testing

=====

firstly lets complete only functional dev + QA testing
take to PROD.. and later on write unittesting cases.

====

For big projects.. lot of code & project running for many years

- Unit Testing MANDATORY

small - medium project

- currently unittesting might be optional. But as the time passes, you have to mandatory implement unit testing in system.

Mar 2025

Dev + QA

2027

new requirement change come to this feature.

new feature comes which affects current feature 2025.

existing functionality can break.

Unittesting

=> regardless of which employee has developed / tested feature, even in future, if we change the requirement, with unittesting we can be sure, that we are not breaking existing system.

every 3months - 1 GOLIVE
every 6months - 1 GOLIVE

AGILE

every 2 weeks finish features.. & deploy to PROD also (golive)

Sprint1: 10 features
S2: 6 features

S3: 5 features

Dev + QA

With new features, our existing system should always be stable. It should not break.

==========

Unittesting is mandatory

You write additional code which checks your functional code.

CODE protects CODE

THis additional unit-testing code, you keep always running in system, for entire future.
And whenever existing functionality is broken, unittesting can identify that.
Once we identify the issue, devs are supposed to fix it.

==========

If functional dev - 2 days
Unittesting - 1 day
2 days for unittesting
3 days also required

=> 3, 4, 5 - dev

AI

100 lines of functional code
all unittesting is implemented.
---

for new feature, we do coding
5lines - 10lines

Unittesting mandatory
we might break old code when we write new code.
manually verifying is not possible.
so we need unittesting
code(unittesting) which protects code(functiona)
immediately, or sometime in future, since it take more time to complete the feature & take to PROD.

"unit" testing
every possibility of code is tested.
don't define it based on how many test methods you write while testing 1 functional method.
consider that each possibility of functional code is tested.

While doing unittesting:

- for every functional class, you write 1 unittesting class
- You unit-test functional code of ONLY that class, not outside class

Unittesting + Mocking + CodeCoverage

- functional code: src/main/java
  test code: src/test/java
- You write unittesting for classes where some business logic exist.

src/test/java
com.hulkhiretech.payments.service.impl.PaymentServiceImplTest

JUnit (Unittesting) - Mockito (Mocking)
TestNG - PowerMokito

Testing:
You execute functionality & check whether its working as expected.

expect result:
execute your functionality
actual result:

if expected is same as actual

In your unittest class
get object of PaymentServiceImpl & invoke the desired method.

======

```
@Slf4j
public class PaymentServiceImplTest {

	@Test
	public void testCreatePayment() {
		log.info("Test case for createPayment method");
	}

}
```

when writing pure unit-testing cases, the spring container is not activated, not initlized.. so we cannot get any spring beans for doing any action.

we need to learn on the junit framework operates.

1. to write test methods: @Test

- how to get object reference of the functional class that you want to invoke??

@ExtendWith(MockitoExtension.class)
@InjectMocks
for the functional class field.
    whichever java class you want to write unittesting for.

Mock means dummy

@Mock

How to write testMethod:

Arrange
- Arrange the required data to call the functional method

Act
- Call the functional method

Assert
- Check the outcome "expected vs actual"

"testMethod executed success"
"testcase is success"
"unittest passed"

@Mock
private PaymentStatusService paymentStatusService;

@InjectMocks
private PaymentServiceImpl paymentServiceImpl;

Here Mockito framework creates dummy object of PaymentStatusService, and gives reference of that object to PaymentServiceImpl.
This @Mock provides dummy object, however if you call methods on that object, it will not be executed. It just bypasses & give default return type.

---
when unittesting of existing system fails, you need to first analyze the impact of past system & fix that impact, and then fix the unit-test case

Code Coverage.
- how much functional code did you cover with your unit-testing.
- Your code coverage percentage to be above 80%

TransactionDTO txnDto = transactionDao.getTransactionByReference(txnReference);

this transactionDao is mock object. so it never calls the getTransactionByReference & always the default return will be null.

We need to change the default return & give some txnDTO object, so we can test rest of the logic

arrange:
TransactionDTO txnDto = new TransactionDTO();
when(transactionDao.getTransactionByReference(txnRef)).thenReturn(txnDto);