Next Week ≈ 3rd Oct

→ for the people who have solved Assg
= (all 7 APIs working perfectly)

→ groups of 4

→ each team will have to implement a project → 4 diff services

→ 3 weeks

→ Micro services

→ Logging and Monitoring

→ LB / API G/W

(1)

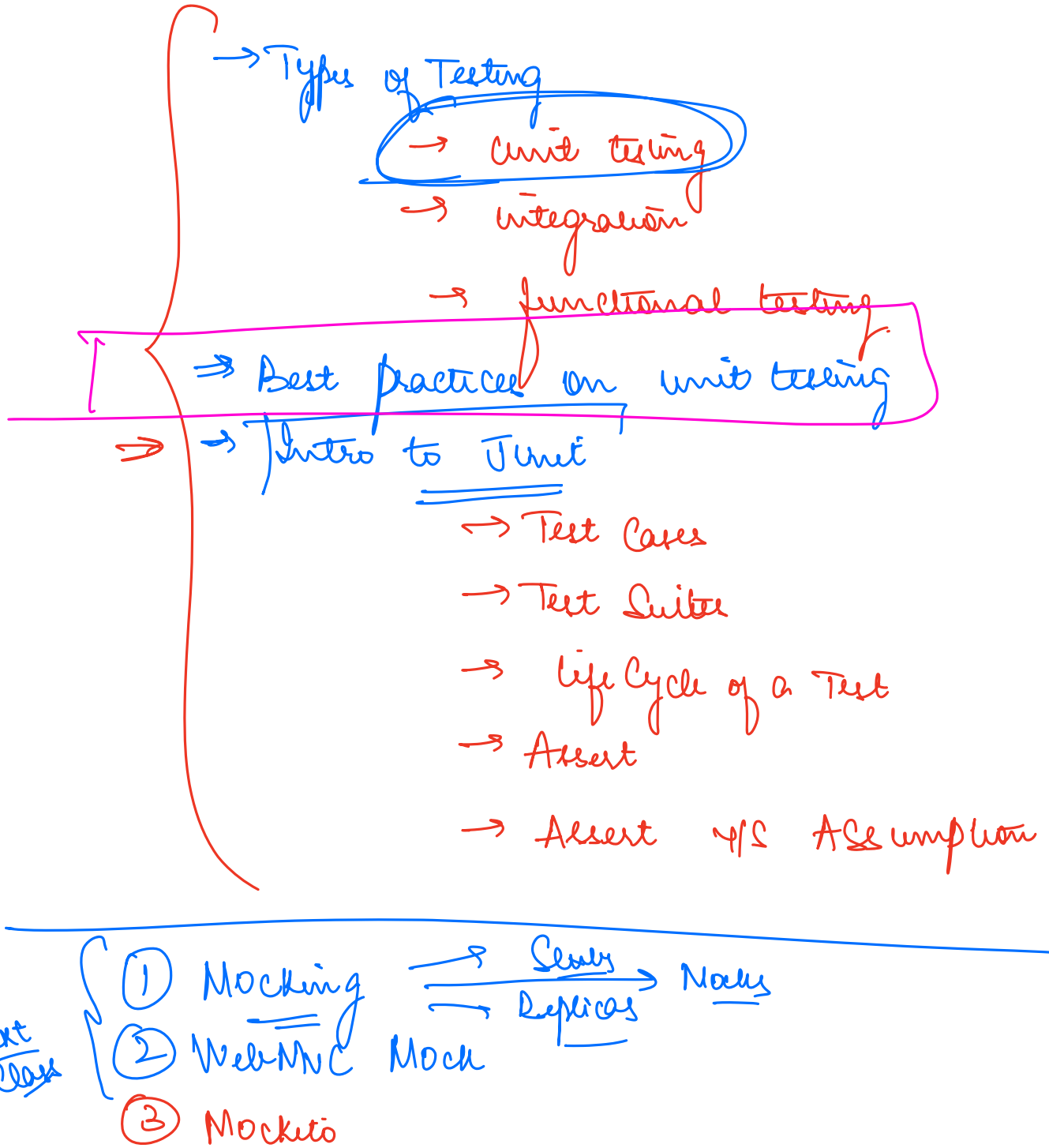
→ How to mention these in resume

(1)

→ 20% =

→ 40% =

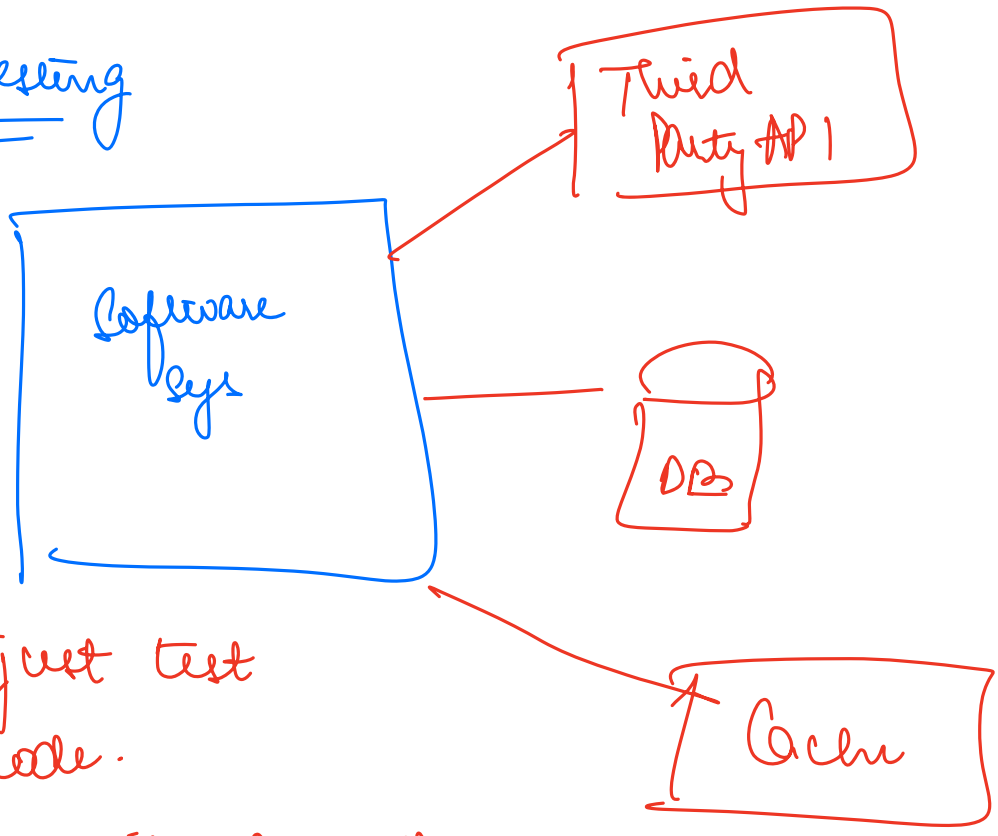
Unit Testing



Next Class

1. Mocking → Stubs / Replicas → Mocks
2. WebMVC Mock
3. Mockito

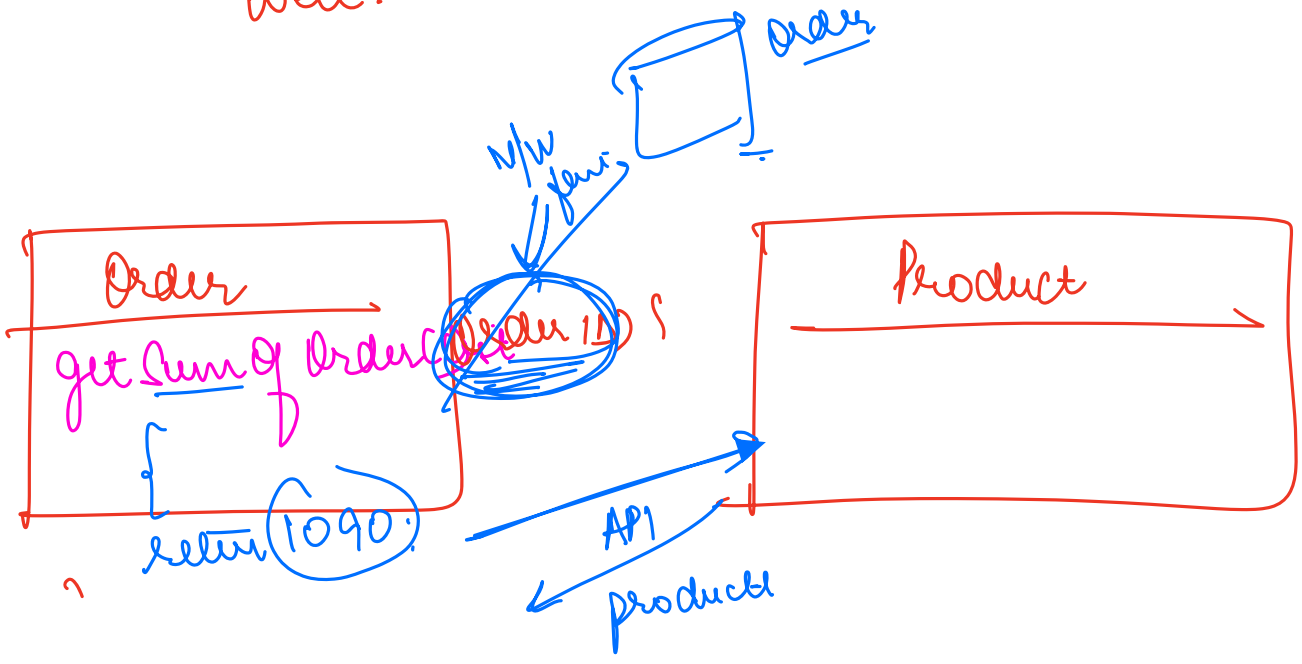
1) Integration Testing



→ Don't just test your code.

→ Also test if a code is working fine with a 3rd party System.

→ invoke the 3rd party systems as well.



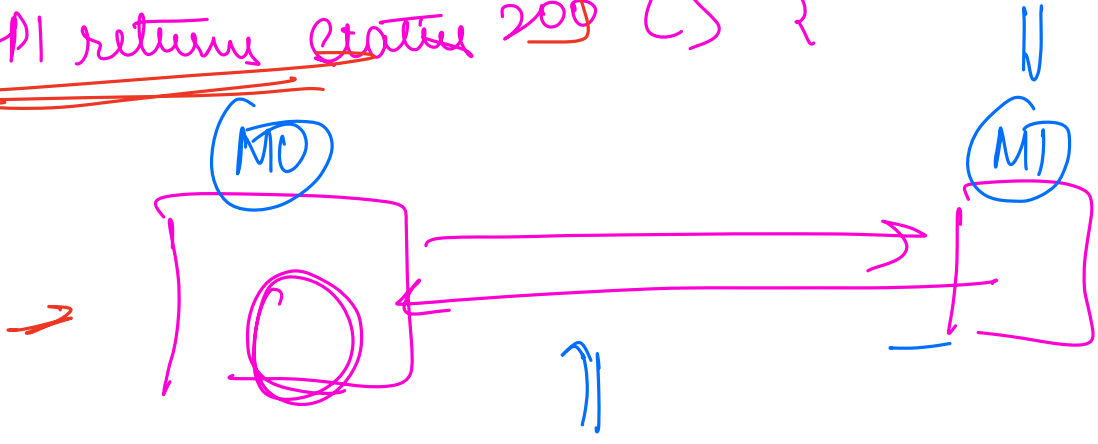
Integration test

→ check if sum of order 1235 is 1090

(1) testing where 3rd party Cys will be called

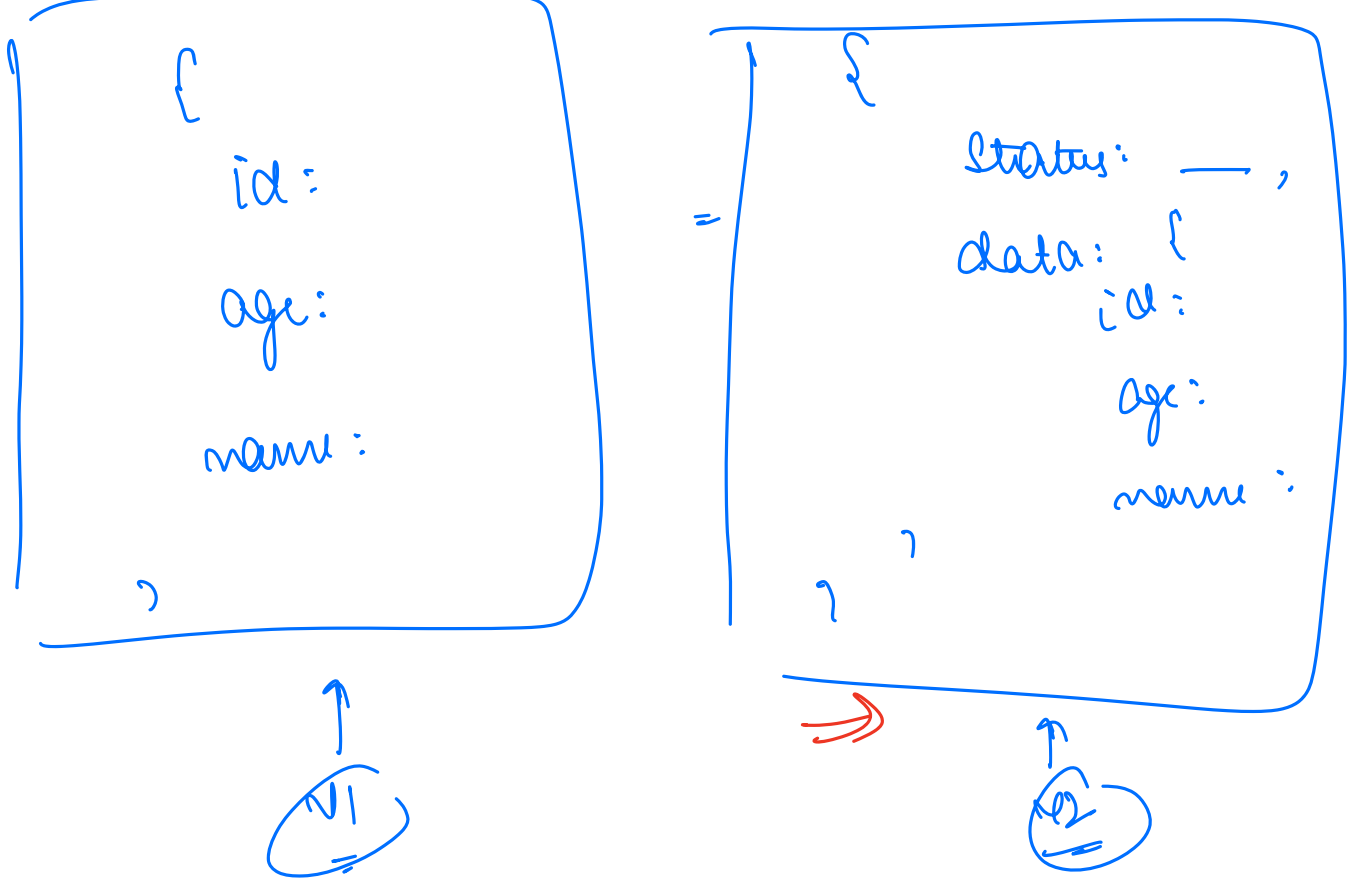
(2) testing 3rd party APIs

test API returns status 200 (S) {

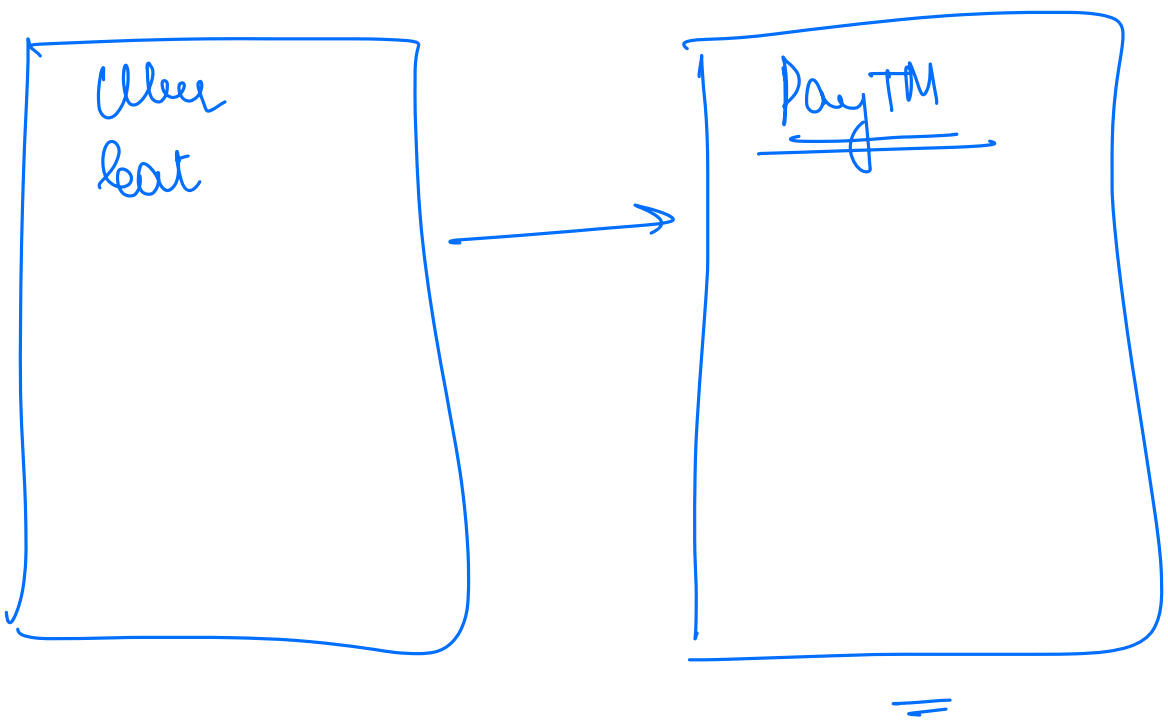


}

3



→ flakiness



{

Status : FAILURE, ✗

{ SUCCESS-ON,

SUCCESS_TRANSFR_TO-BAN

→ FAIL_OTP,

?

if (api. status != failure || != FAIL-OTP) {

→ // Success order. ⇒

Unit Tests

→ typically test a small unit (code block) of your App:

→ Single class / method.

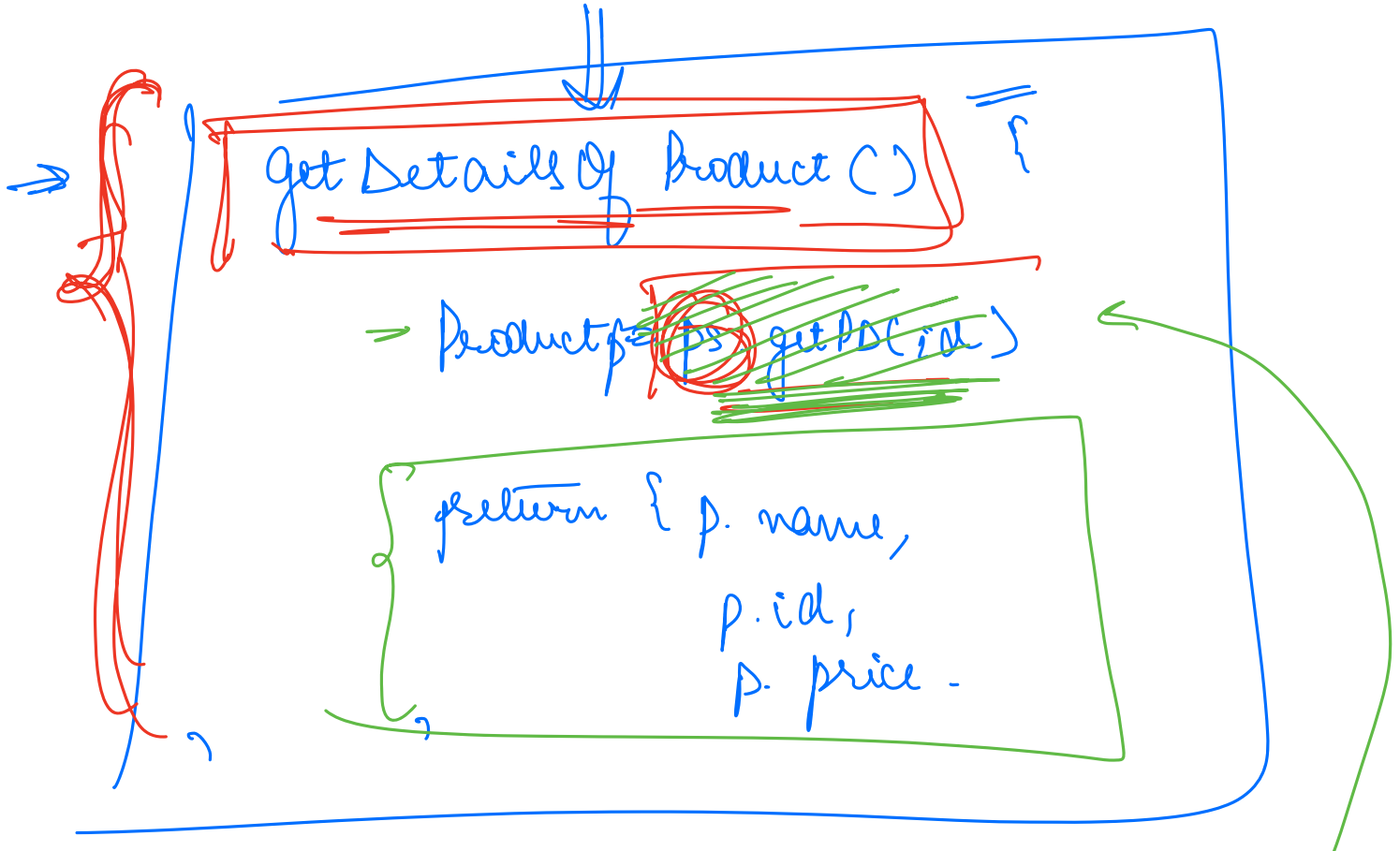
→ if that class or method is doing its responsibilities correctly.

→ short and fast.

→ not dependent on external dependencies

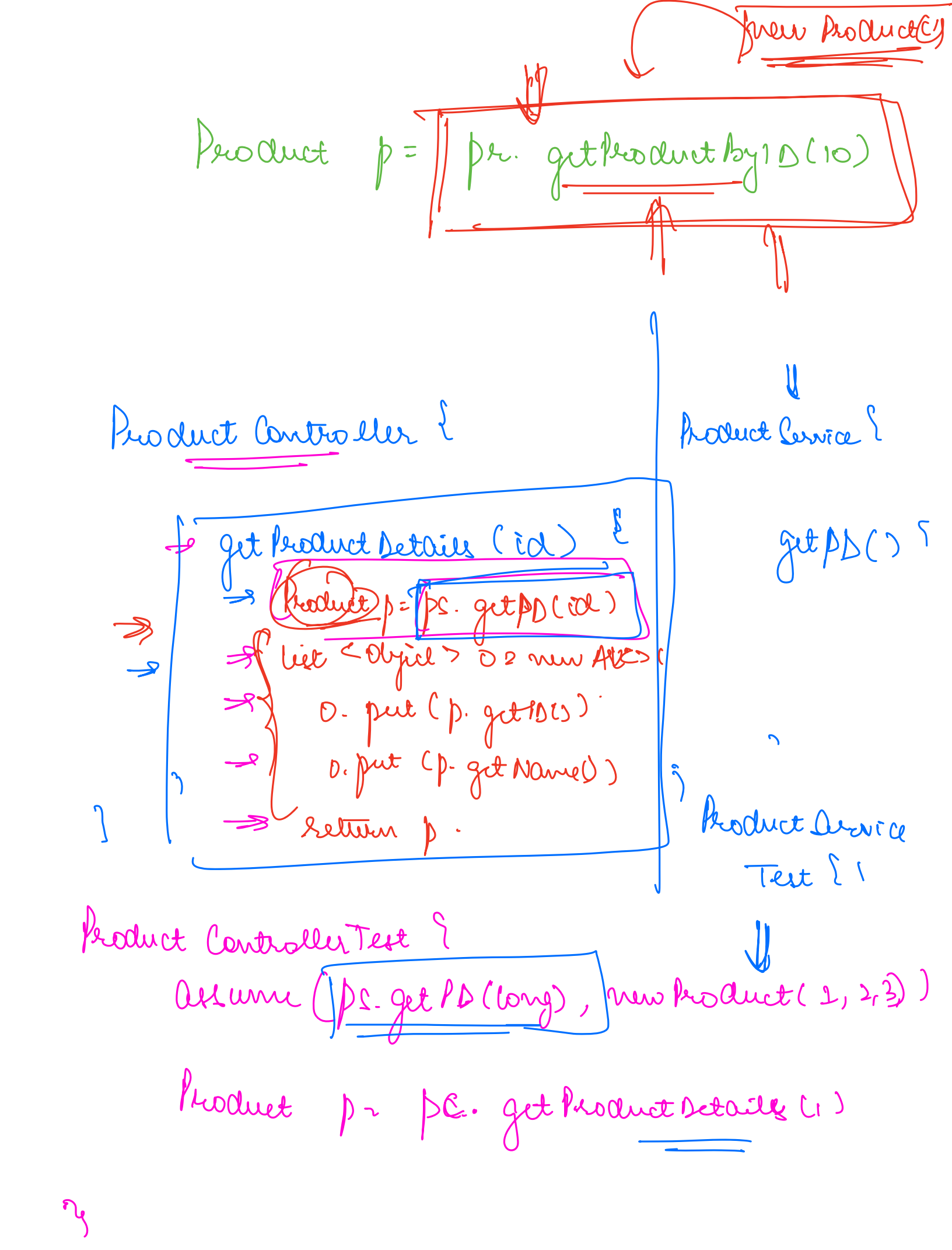
Code Coverage ⇒ what % of your codebase is being tested via one or other unit test.

90%



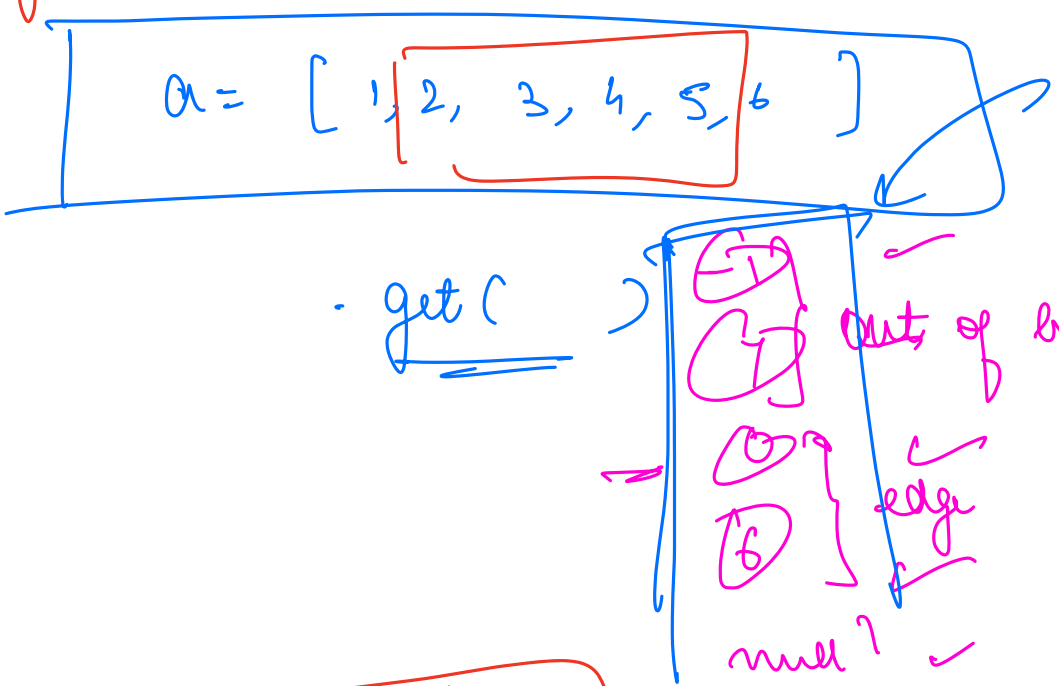
Mocking

ps.getPD(1) ⇒ new Product (———);


new Product()
Product p = pr. getProductById(10)
Product Controller {
Product Service {
getPD() {
getProductDetails(id) {
Product p = PS. getPD(id)
List<Object> O = new ArrayList<>();
O. put(p. getID());
O. put(p. getName());
return p.
Product Service Test {
ProductControllerTest {
Assume(PS. getPD(long), new Product(1, 2, 3))
Product p = PC. getProductDetails(1)

→ testing that code in isolation assuming everything else is working fine.

→ for each method in your codebase, you write multiple test cases.



→ a.get(2) → 6

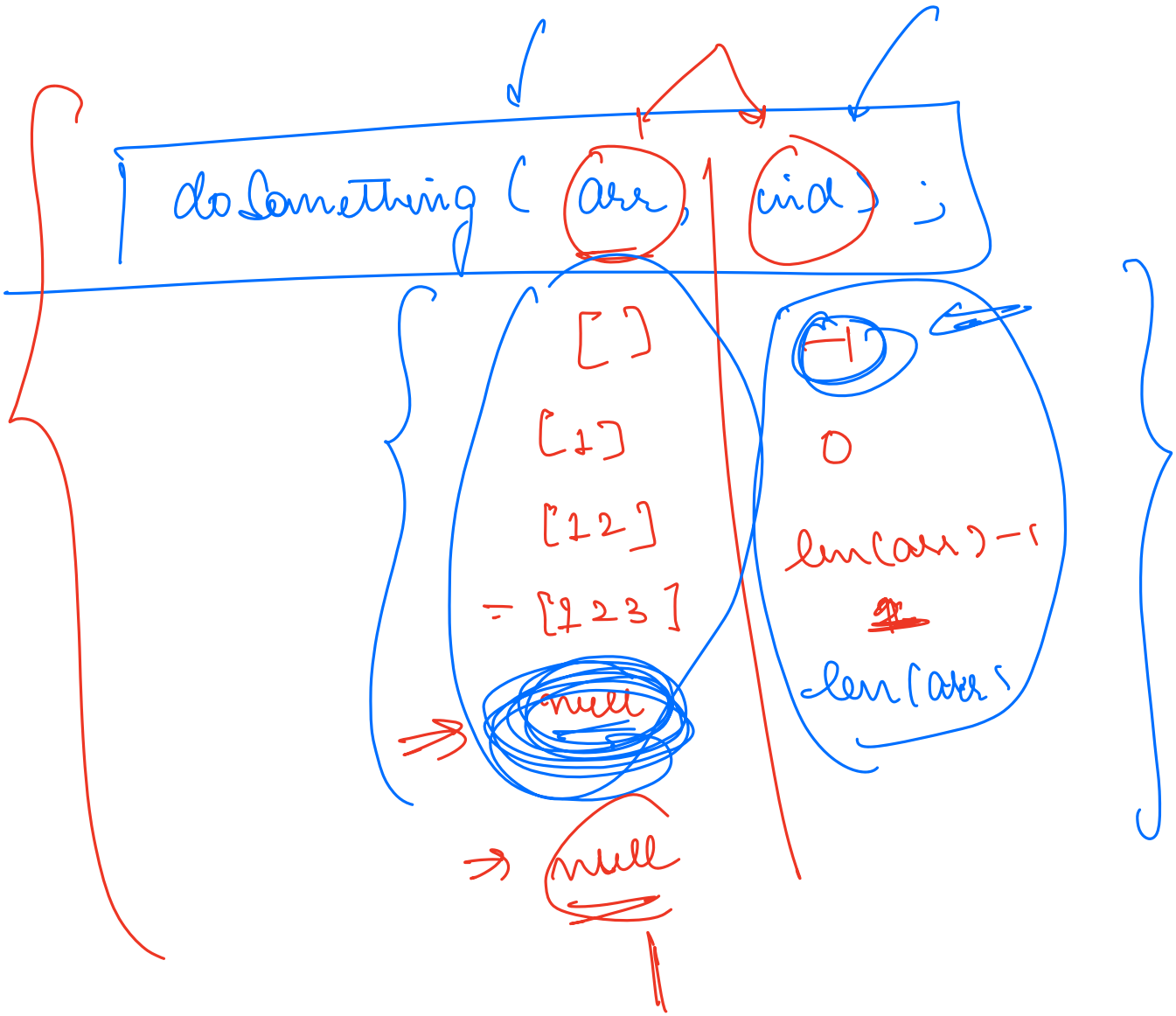
→ ① bad cases ⇒ not expected situation
→ array index out of bound
→ null.

② corner cases ⇒ situation where typical errors can happen

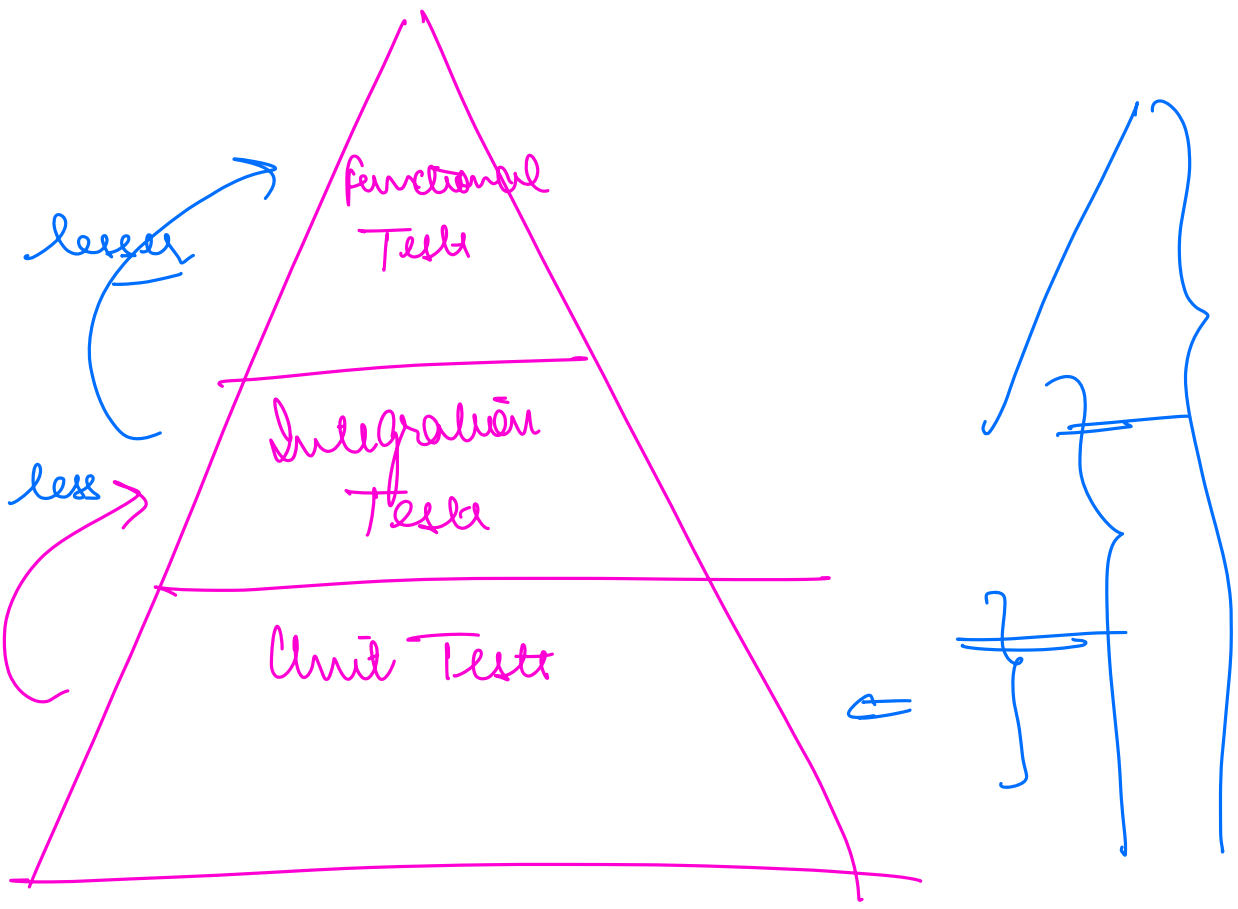
⇒ 0th index

⇒ last index

③ happy cases → situations that should go right



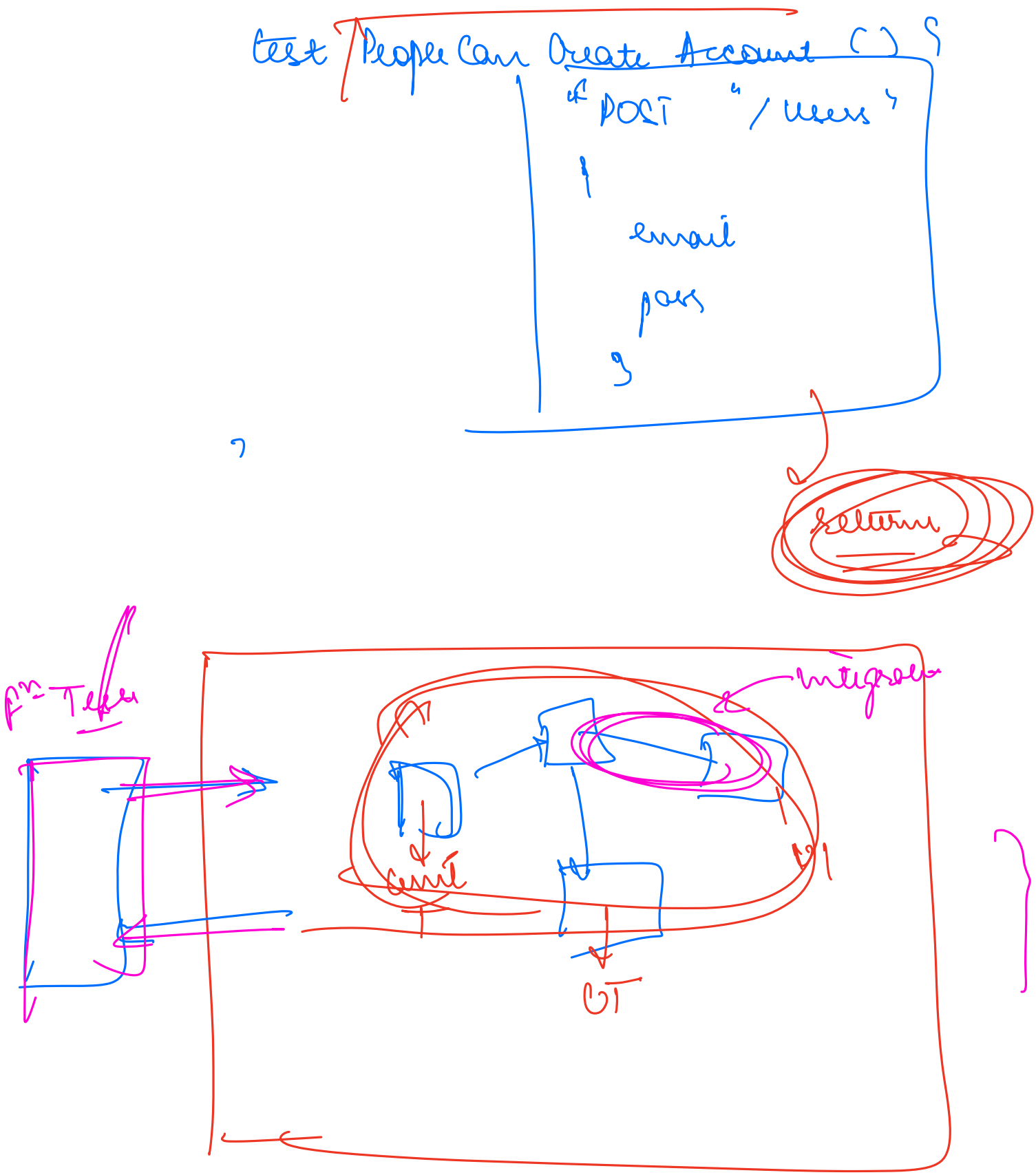
Functional Tests

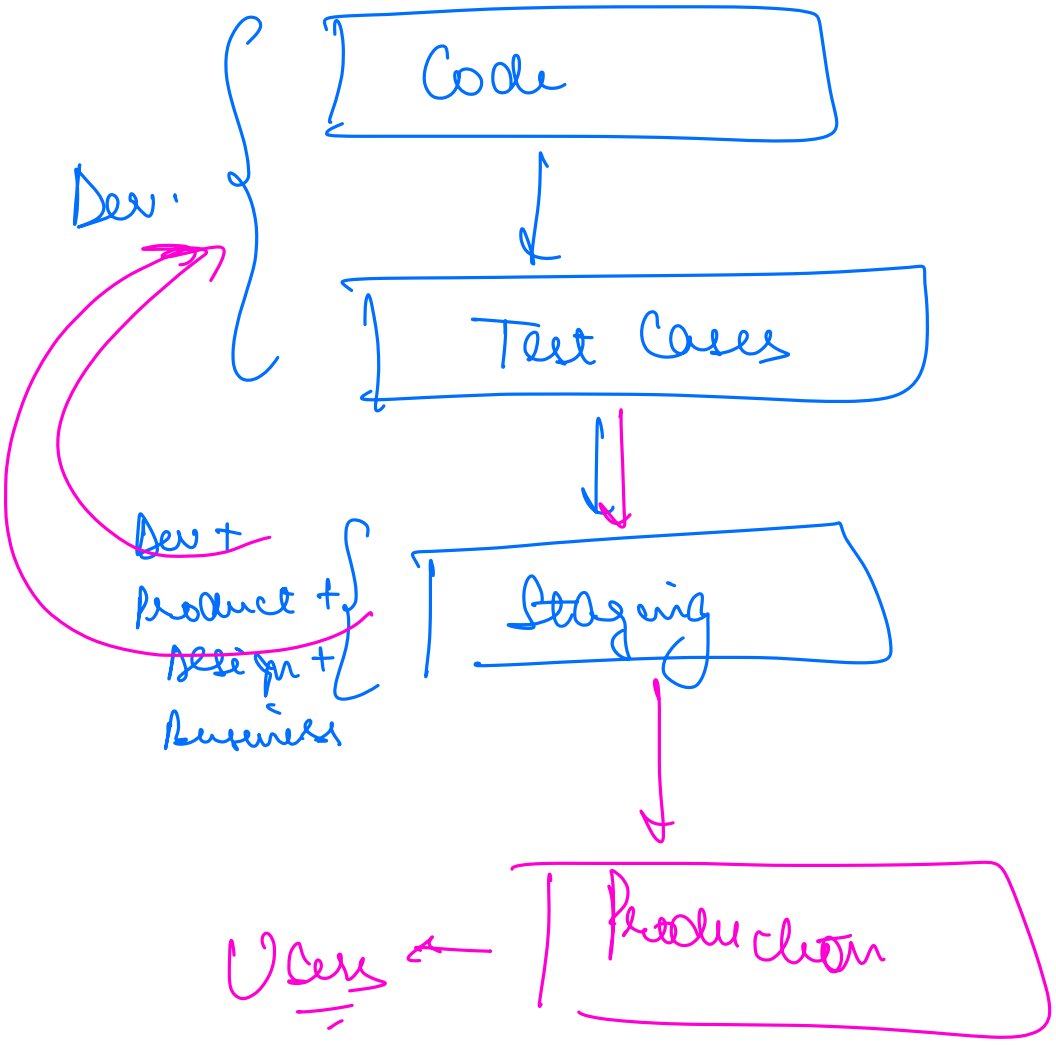


Functional Tests are just testing business req.

↳ end to end use cases.

(1) People should be able to create Account


test People Can Create Account ( ) {
" POST " / Users ",
{
email
pass
}
}
Returns
Pre-Test
Integration
Unit
UI
UT



Break till 10:27

Unit Testing Best Practices

1. Fast

Arrange
Act
Assert

Create obj, mock depend

create obj of classes whose methods you will be testing.

Call the thing that you have to test and store its result

Check if the call returned what you expected

C → Creat
c → Call
C → Check

Calculator {

Calculate (int a, int b, Op = 0) {

}

```
Calculator c = new Calculator();
int d = c.calculate(1, 2, ADD);
Assert(3, d);
```

(2) Isolated

→ mock every external dependency

(3) Repeatable

→ no flakiness

(4) Self Checking / completely automatic.

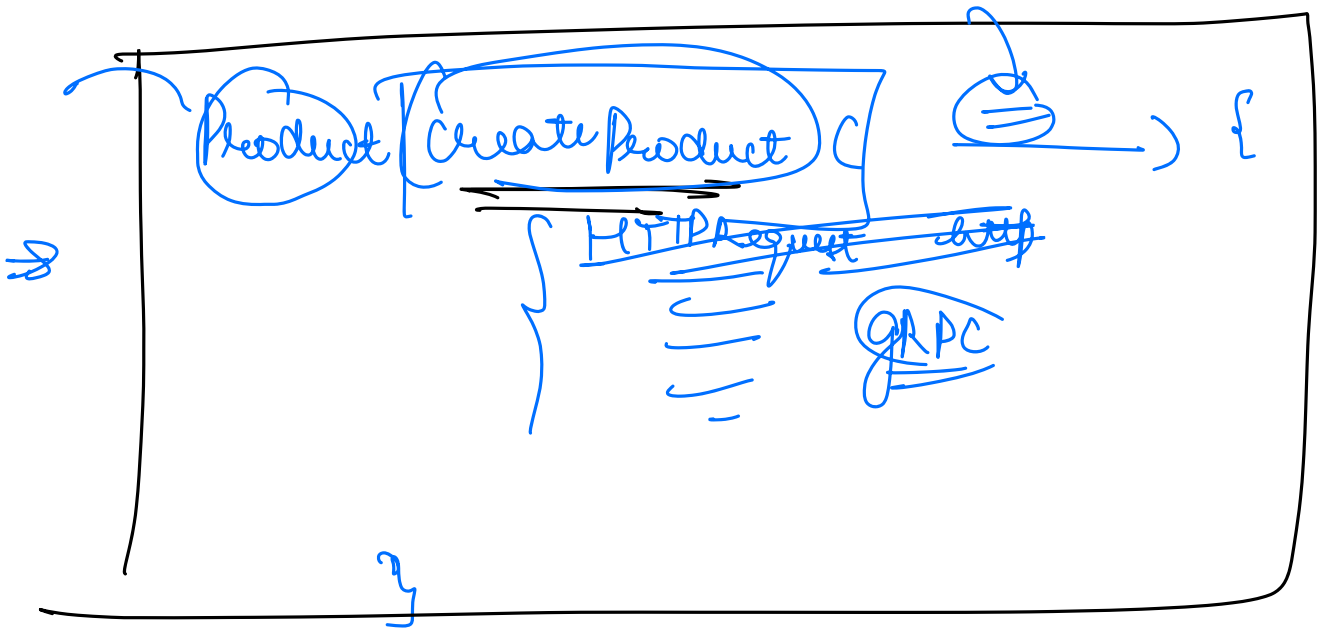
→ shouldn't req any human input

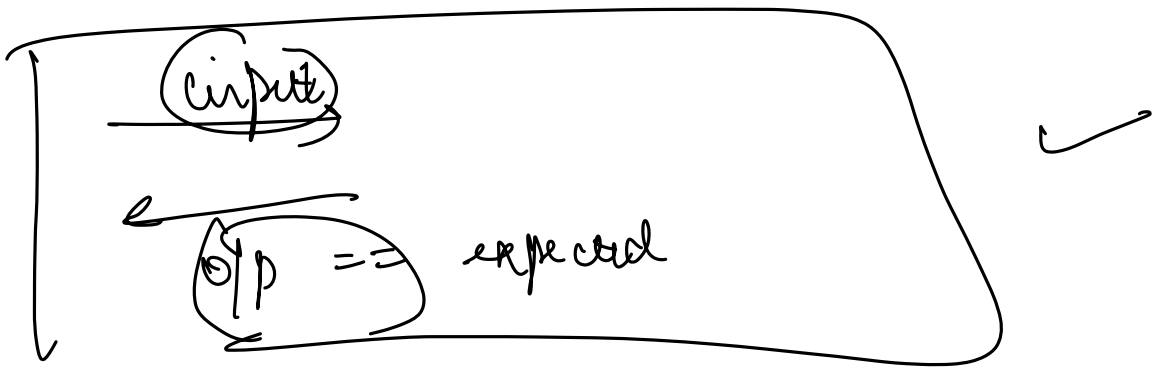
```
Scanner sc = new Scanner();
```

(5) Test behaviour, not implementation

→ test of the behaviour of the unit under test is what you are expecting.

→ don't test how the behaviour is happening.





⇒ dont test if a particular way of achieving o/p is there

eg. dont test if a particular lib is being used.

```
Product Controller Test {
    - Create Product Test ( ) {
        Product Service
```

Test cases allow a dev to take the role of a client.