Agenda

1) What to test ?
2) Best practices for UNIT Testing
3) Mocking
   - Type of doubles

UNIT Testing

- Testing the code unit in Isolation
- Test logic of code unit

1) What to test

String

```
Calculator {
    int add(int a, int b)
    {
        return a+b;
    }
}
```

① EDGE Cases ⇒ Corner Cases

(Test cases which are not easy to get)

② -Ve Test Cases ⇒

(Test case for which our code will generate wrong output)

③ +Ve Test Cases ⇒ Happy path

eg ① 1+1 = 2 ⎫
⎬ +Ve
eg ② -1+(-1) = -2 ⎭

eg ③ 1+0 = 1 ⎫ Edge Case

eg ④ 0+0 = 0 ⎫ edge case

eg ⑤ 1 + Int_Max ⎫ edge case | -ve

eg ⑥ 1+'a' → type error -Ve scenario

eg ⑦ True+1 → -ve

eg ⑧ [1,2]+3 → -ve

eg

Object a = readLine(←)

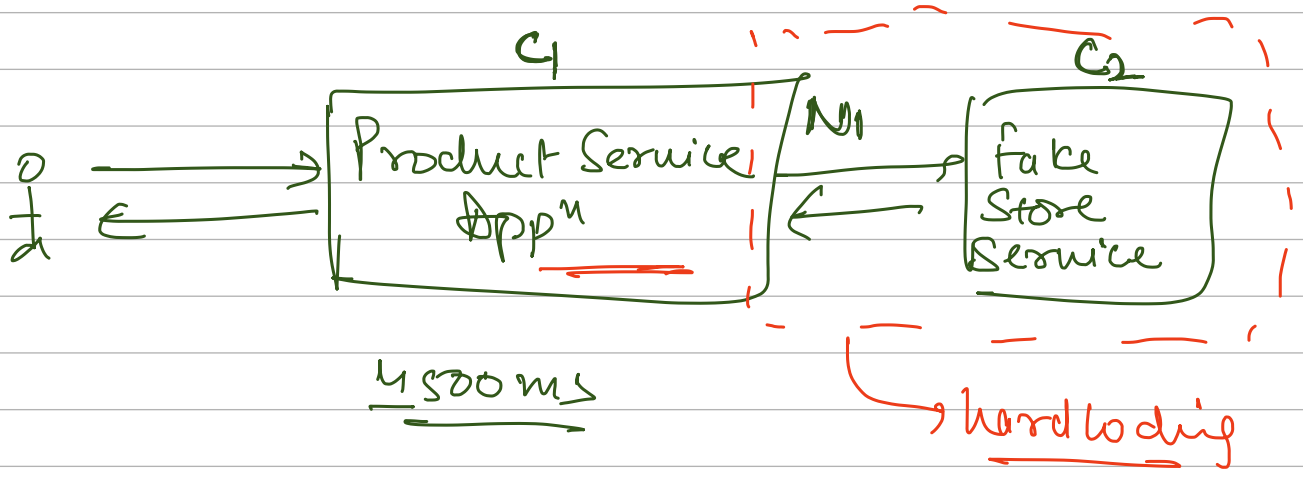
↓

Result = add((int)a, (int)b)

Properties of UNIT testing

① Fast

ProductController {
createProduct
} → FakeStoreService



500 ms

$t = C_1 + N_1 + C_2$

test time = C_1

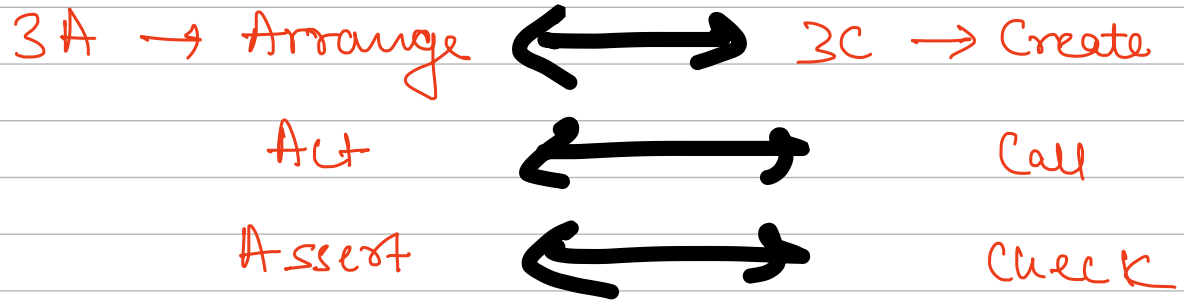
$C_1 << C_1 + N_1 + C_2$

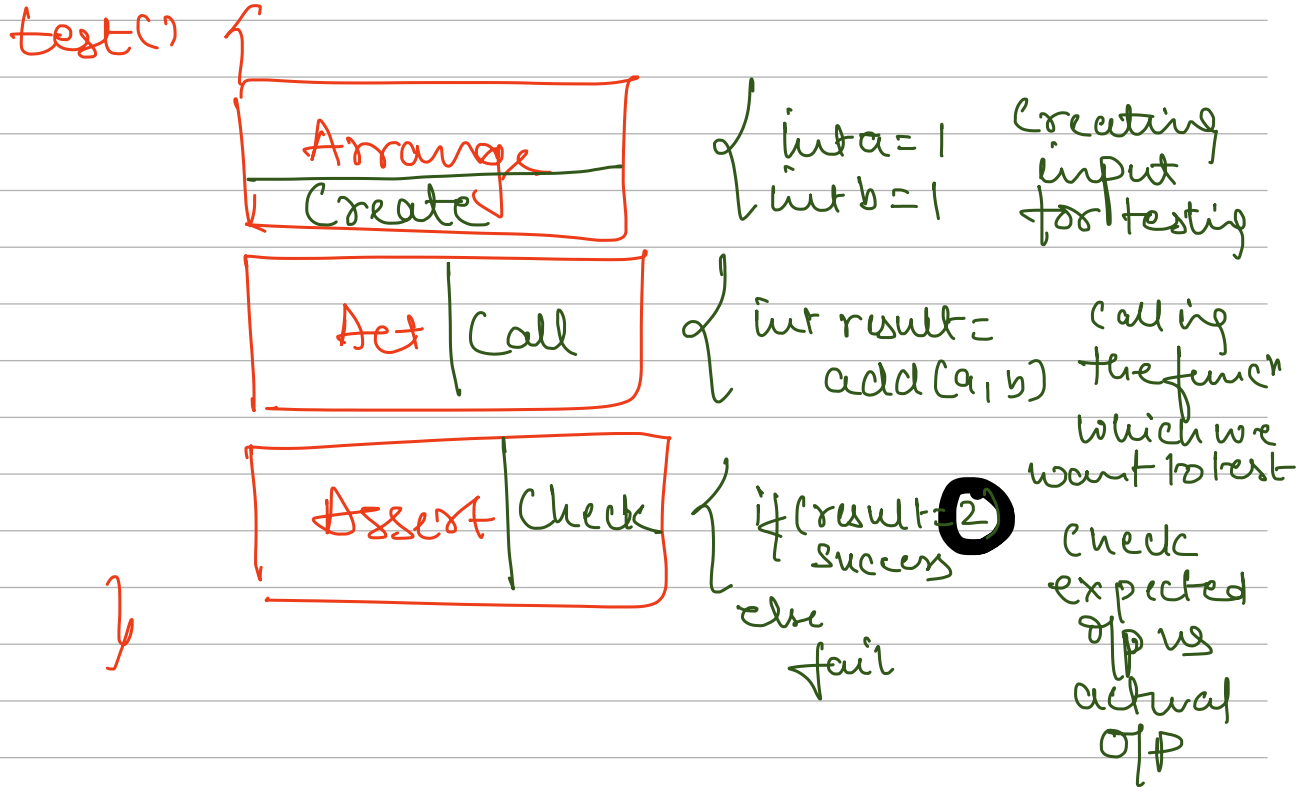
1 μs μs + ms + μs

μs << ms

② 3A's framework

3C's framework





⇒ Always hardcode expected o/p as we don't it to be changed

③ Isolation

⇒ Success / failure of testcase should depend only on the business logic (behavior) of function, NOT on the dependency

⇒ Ideally we should **Correct Code** dependencies

MOCKING

④ Repeatable

⇒ Our testcase should return the same output for same input NO matter how many times we run it.

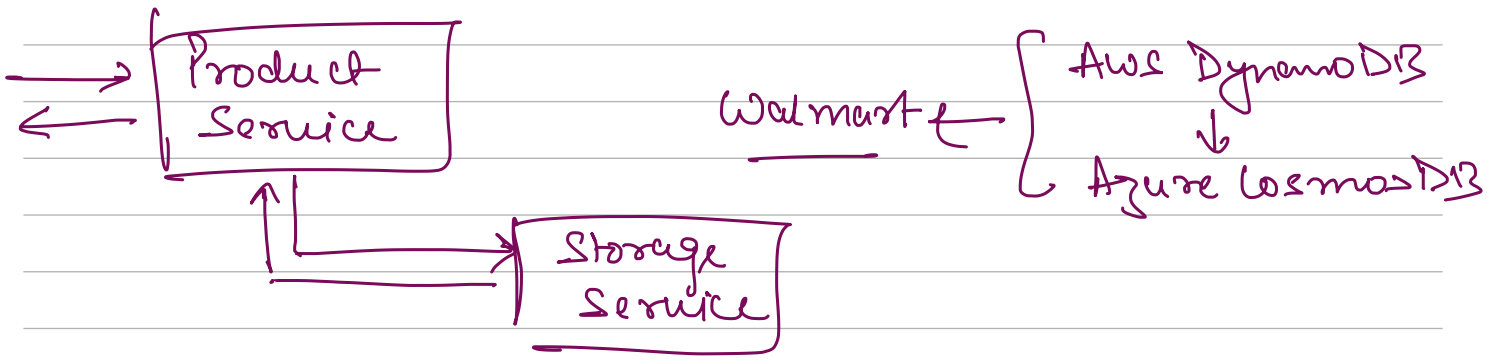
⇒ TC should Never be flaky

⑤ Self-Checking

To run any TC, No Human intervention should be required

eg
```
testAdd {
    int a = 1
    int b = 1
    int result = add(a,b)
    SystemOutPrint(result);
}
```

(6) Test behaviors, Dont test implementation



eg

```
multiply ( int a, int b)
{
    return a*b;
}
```

Impl (1)

```
multiply ( int a, int b)
{   int result;
    for ( 0 to a)
    {   result = result + b;
    }
    return result;
}
```

Impl (2)

multiply (2, 3) = 6

⇒ Because impl can change over time, as long as behavior does not change TC should not require any change.

⇒ We should always test behavior NOT impl.

eg Test save data in DB

↳ DB connection → MySQL

Repo {

DB Conn = ✓ DI

save (———)

}

Test save {

if(Repo.dbconn is Instence MySQL conn)

}

ProductController Test {

@Mock

ProductService productSerive;

@Autowired

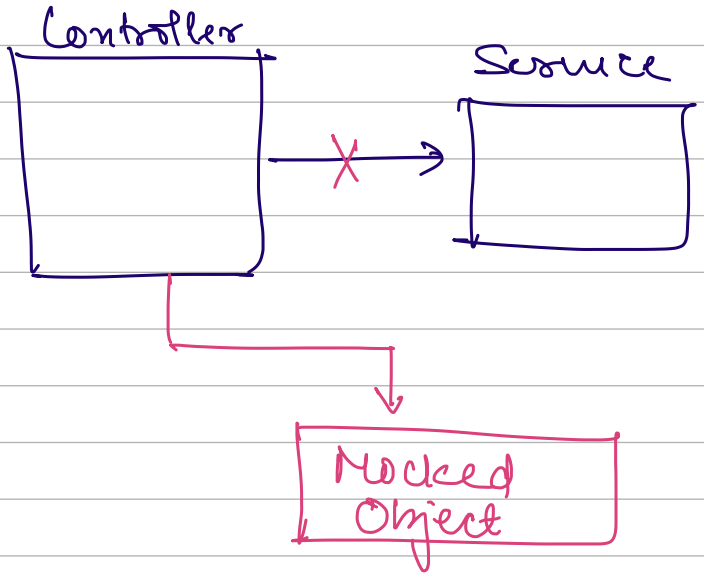
ProductController pc;

```
testGetProductById() {

    // Arrange
    int id=1;
    Product product = new Product();
      product.setId(1);
      product.setTitle("Macbook");
    when(ProductService.getProductById(id))
      .thenReturn(product);

    // Act
    Product p = pc.getProductById(id);

    // Assert
    if (p.getTitle.equals("Macbook")) {
        Success
    } else {
        fail
    }
}
```

pc.getProduct
↓
PS.getProduct

Mocking

- Isolated
- Repeatable



In Unit test cases to test func^n in isolation we mock the dependencies

Mocked dependencies needs to be told about behaviour

↳ O/P when will be called

↳ Hard Coded

Doubles will help achieving mocking

Type of Doubles

↳ MOCK

↳ STUB

↳ FAKE

MOCK

A double where we just return a value

[when(ps.getProductById(1)).return(p);]

[ps.getProductById(2);]

[Not dynamism available]

Scenario

1. Create 5 products
2. Get the count of products ⇒ 5
3. Create 1 more product
4. Get the count of products ⇒ 6

→ when(ps.getProduct(count)).return(5)

2 ⇒ 5
4 ⇒ 5] X

② STUB

↳ A class we create to replicate the Behavior of actual service

Test Package

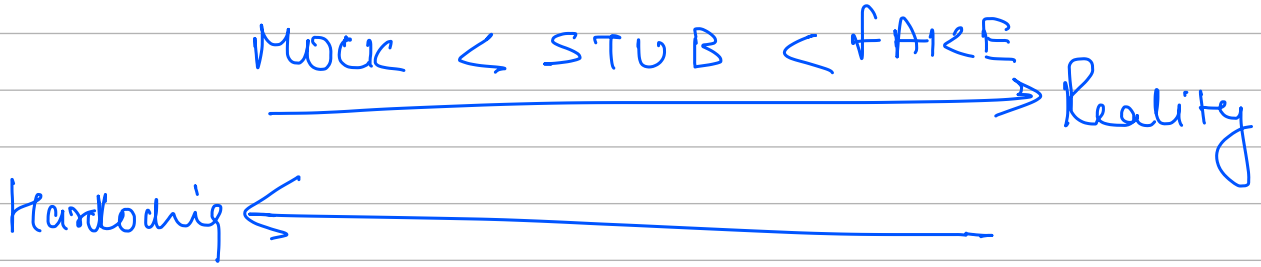
eg
```
ProductServiceStub {
    int count=0;
    void createProduct(Product P)
    {
        count++;
    }
    int getProductCount()
    {
        return count;
    }
}
```

Scenario
(5) Get a Product Id=3 and check the contains

FAKE
→ Moving closer to realstic Impl

```
Class ProductServiceFake implent ProductService{
    HashMap<Integer, Product> inMemoryDB
        =new HashMap<>();
    int id=0;
    CreateProduct(Product p)
    {
        p.set(id++);
        map.put(id, p);
    }
}
```


psf
Test PC 2


MOCK < STUB < FAKE
Reality
Hardcoding