Name : CHANDRU A

College : Kongu Engineering Colleges

**EXERCISE 1: MOCKING AND STUBBING**

This project demonstrates mocking and stubbing using Mockito, focused on a GradingApi interface that returns academic data such as student names, individual subject marks, grade calculations, and attendance. The StudentService logic is tested in isolation without using the actual API.

**Objectives:**

Mock External Dependencies

Simulate the behavior of the GradingApi interface using Mockito.

**Stub Return Values**

Use when(...).thenReturn(...) to mimic API responses for marks, grades, and attendance.

**Validate Service Logic**

Ensure StudentService logic behaves as expected with mocked data.

**Verify Method Calls**

Confirm interaction with API methods (e.g., recordAttendance()) using verify(...).

**Code & Output:**

**GradingApi.java**

package org.example;

public interface GradingApi {

int getMarks(String studentId, String subject);

String getGrade(String studentId);

boolean isPassed(String studentId);

double getAttendancePercentage(String studentId);

void recordAttendance(String studentId);  
}

**StudentService.java**

package org.example;

public class StudentService {

private final GradingApi api;

public StudentService(GradingApi api) {

this.api = api;

}

public int getSubjectMarks(String id, String subject) {

return api.getMarks(id, subject);

}

public String getStudentGrade(String id) {

return api.getGrade(id);

}

public boolean hasPassed(String id) {

return api.isPassed(id);

}

public double getAttendance(String id) {

return api.getAttendancePercentage(id);

}

public void markAttendance(String id) {

api.recordAttendance(id);

}

}

**StudentServiceTest.java**

package org.example;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

public class StudentServiceTest {

@Test

public void testGetMarks() {

GradingApi api = mock(GradingApi.class);

when(api.getMarks("S101", "Math")).thenReturn(85);

StudentService service = new StudentService(api);

int marks = service.getSubjectMarks("S101", "Math");

System.out.println("Test 1: getMarks(S101, Math) = " + marks);

assertEquals(85, marks);

}

@Test

public void testGetGrade() {

GradingApi api = mock(GradingApi.class);

when(api.getGrade("S102")).thenReturn("B");

StudentService service = new StudentService(api);

String grade = service.getStudentGrade("S102");

System.out.println("Test 2: getGrade(S102) = " + grade);

assertEquals("B", grade);

}

@Test

public void testPassStatus() {

GradingApi api = mock(GradingApi.class);

when(api.isPassed("S103")).thenReturn(true);

StudentService service = new StudentService(api);

boolean passed = service.hasPassed("S103");

System.out.println("Test 3: isPassed(S103) = " + passed);

assertTrue(passed);

}

@Test

public void testAttendancePercentage() {

GradingApi api = mock(GradingApi.class);

when(api.getAttendancePercentage("S104")).thenReturn(92.5);

StudentService service = new StudentService(api);

double attendance = service.getAttendance("S104");

System.out.println("Test 4: getAttendance(S104) = " + attendance);

assertEquals(92.5, attendance);

}

@Test

public void testRecordAttendanceVerification() {

GradingApi api = mock(GradingApi.class);

StudentService service = new StudentService(api);

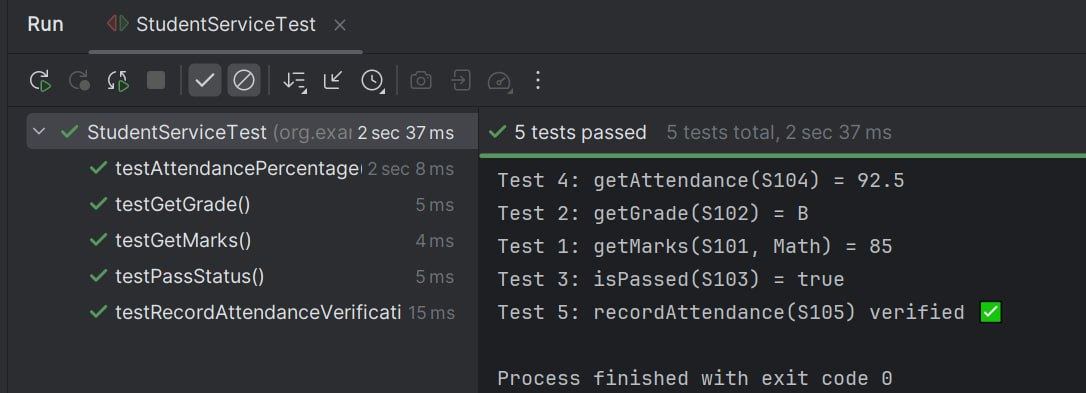
service.markAttendance("S105");

verify(api).recordAttendance("S105");

System.out.println("Test 5: recordAttendance(S105) verified ✅");

}

}



Output:

All tests validate logic of StudentService using mocked responses. No real database or backend is needed. This approach provides:

Fast & isolated unit testing

Clear validation of business rules

**EXERCISE 2: VERIFYING INTERACTIONS**

**Scenario:**

This test verifies that DietService correctly interacts with its dependency DietTrackerApi by calling methods like logMeal() and submitPlan(). Mockito is used to mock the API and verify method invocations with accurate arguments.

**Objectives:**

Mock Dependencies

Replace the real DietTrackerApi with a mock to isolate the service logic.

**Trigger Action**

Call logMeal() and finalizePlan() to simulate diet logging.

Verify Interactions

**Use verify() to confirm:**

logMeal("Oats", 200) was called correctly.

submitPlan() was called exactly once.

Some meals were never logged.

**Code & Output:**

**DietTrackerApi.java**

package org.example;

public interface DietTrackerApi {

void logMeal(String foodItem, int calories);

String submitPlan();  
}

**DietService.java**

package org.example;

public class DietService {

private final DietTrackerApi api;

public DietService(DietTrackerApi api) {

this.api = api;

}

public void addMeal(String item, int calories) {

api.logMeal(item, calories);

}

public String finalizePlan() {

return api.submitPlan();

}  
}

**DietServiceTest.java**

package org.example;

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

public class DietServiceTest {

@Test

public void testLogMeal\_Oats() {

DietTrackerApi mockApi = mock(DietTrackerApi.class);

DietService service = new DietService(mockApi);

service.addMeal("Oats", 200);

System.out.println("testLogMeal\_Oats: Verifying Oats (200 cal) logged");

verify(mockApi).logMeal("Oats", 200);

}

@Test

public void testLogMeal\_Twice() {

DietTrackerApi mockApi = mock(DietTrackerApi.class);

DietService service = new DietService(mockApi);

service.addMeal("Banana", 100);

service.addMeal("Banana", 100);

System.out.println("testLogMeal\_Twice: Banana (100 cal) logged twice");

verify(mockApi, times(2)).logMeal("Banana", 100);

}

@Test

public void testLogMeal\_NeverCalledForSweets() {

DietTrackerApi mockApi = mock(DietTrackerApi.class);

DietService service = new DietService(mockApi);

service.addMeal("Salad", 80);

System.out.println("testLogMeal\_NeverCalledForSweets: Sweets should not be logged");

verify(mockApi, never()).logMeal("Sweets", 300);

}

@Test

public void testFinalizePlanCalledOnce() {

DietTrackerApi mockApi = mock(DietTrackerApi.class);

DietService service = new DietService(mockApi);

service.finalizePlan();

System.out.println("testFinalizePlanCalledOnce: submitPlan() called");

verify(mockApi, times(1)).submitPlan();

}

@Test

public void testFinalizePlan\_ReturnsSuccess() {

DietTrackerApi mockApi = mock(DietTrackerApi.class);

when(mockApi.submitPlan()).thenReturn("Diet Plan Submitted");

DietService service = new DietService(mockApi);

String response = service.finalizePlan();

System.out.println("testFinalizePlan\_ReturnsSuccess: Response = " + response);

assertEquals("Diet Plan Submitted", response);

}

@Test

public void testLogMeal\_NullFoodItem() {

DietTrackerApi mockApi = mock(DietTrackerApi.class);

DietService service = new DietService(mockApi);

service.addMeal(null, 150);

System.out.println("testLogMeal\_NullFoodItem: null item logged");

verify(mockApi).logMeal(null, 150);

}

@Test

public void testSubmitPlan\_NotCalled() {

DietTrackerApi mockApi = mock(DietTrackerApi.class);

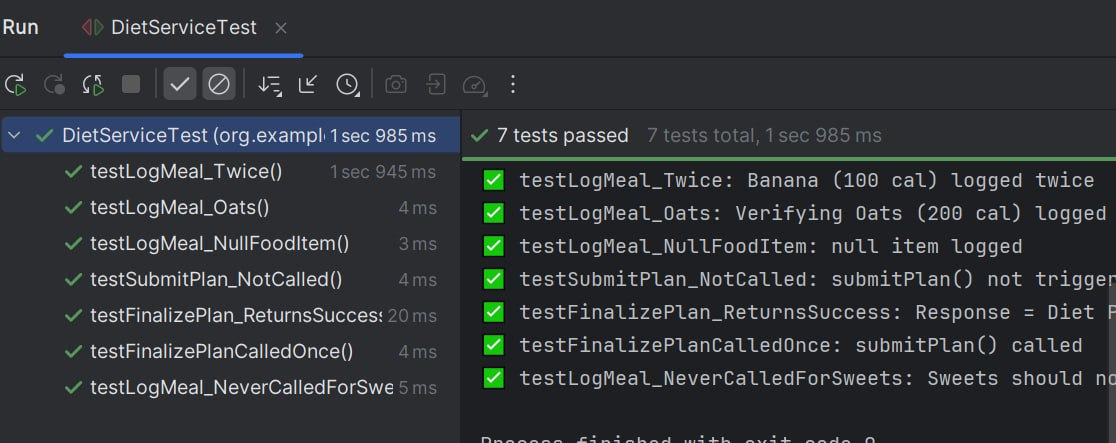
DietService service = new DietService(mockApi);

service.addMeal("Rice", 300);

System.out.println("testSubmitPlan\_NotCalled: submitPlan() not triggered");

verify(mockApi, never()).submitPlan();

}  
}



Output Summary:

Each test ensures:

* Meals are logged with correct calories.
* submitPlan() is triggered only when required.
* Certain meals were never added (e.g., sweets).
* Return value from mocked submitPlan() is validated