**Student Name:** KAMALESHWARAN A

**Registration No:** 22CSR087

**Course/Batch:** KONGU ENGINEERING COLLEGE (B.E COMPUTER SCIENCE AND ENGINEERING)

**EXERCISE 1: MOCKING AND STUBBING**

**Introduction:**

This project uses JUnit and Mockito to test a book recommendation service based on user genre preferences.The service returns genre-specific book suggestions by mocking user preference inputs.

**Objective:**

* To test genre-specific book recommendations returned by the service.
* To use Mockito to simulate user preferences and isolate the logic.
* To ensure the service returns the expected book titles for each genre.

**Implementation Breakdown:**

**UserPreferencesService.java:**

package org.example;

public interface UserPreferencesService {

String getPreferredGenre(String userId);

}

**BookRecommendationService.java:**

public class BookRecommendationService {

private UserPreferencesService preferencesService;

public BookRecommendationService(UserPreferencesService preferencesService) {

this.preferencesService = preferencesService;

}

public String recommendBook(String userId) {

String genre = preferencesService.getPreferredGenre(userId);

switch (genre) {

case "Sci-Fi": return " Recommended: Dune";

case "Fantasy": return "Recommended: Harry Potter";

case "Mystery": return "Recommended: Sherlock Holmes";

case "Thriller": return "Recommended: The Girl with the Dragon Tattoo";

case "Romance": return "Recommended: The Notebook";

case "Historical": return "Recommended: The Book Thief";

case "Biography": return "Recommended: Steve Jobs by Walter Isaacson";

case "Self-Help": return "Recommended: Atomic Habits";

case "Adventure": return "Recommended: The Hobbit";

case "Horror": return "Recommended: It by Stephen King";

case "Philosophy": return "Recommended: The Republic by Plato";

case "Drama": return "Recommended: The Kite Runner";

case "Comedy": return "Recommended: Good Omens";

case "Crime": return "Recommended: Gone Girl";

case "Classic": return "Recommended: Pride and Prejudice";

default: return "Recommended: The Alchemist";

}

}

}

**BookRecommendationServiceTest.java:**

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

public class BookRecommendationServiceTest {

private BookRecommendationService getServiceWithGenre(String userId, String genre) {

UserPreferencesService mockPreferences = mock(UserPreferencesService.class);

when(mockPreferences.getPreferredGenre(userId)).thenReturn(genre);

return new BookRecommendationService(mockPreferences);

}

@Test public void testSciFi() {

assertEquals("Recommended: Dune", getServiceWithGenre("u1", "Sci-Fi").recommendBook("u1"));

}

@Test public void testFantasy() {

assertEquals("Recommended: Harry Potter", getServiceWithGenre("u2", "Fantasy").recommendBook("u2"));

}

@Test public void testMystery() {

assertEquals("Recommended: Sherlock Holmes", getServiceWithGenre("u3", "Mystery").recommendBook("u3"));

}

@Test public void testThriller() {

assertEquals("Recommended: The Girl with the Dragon Tattoo", getServiceWithGenre("u4", "Thriller").recommendBook("u4"));

}

@Test public void testRomance() {

assertEquals("Recommended: The Notebook", getServiceWithGenre("u5", "Romance").recommendBook("u5"));

}

@Test public void testHistorical() {

assertEquals("Recommended: The Book Thief", getServiceWithGenre("u6", "Historical").recommendBook("u6"));

}

@Test public void testBiography() {

assertEquals("Recommended: Steve Jobs by Walter Isaacson", getServiceWithGenre("u7", "Biography").recommendBook("u7"));

}

@Test public void testSelfHelp() {

assertEquals("Recommended: Atomic Habits", getServiceWithGenre("u8", "Self-Help").recommendBook("u8"));

}

@Test public void testAdventure() {

assertEquals("Recommended: The Hobbit", getServiceWithGenre("u9", "Adventure").recommendBook("u9"));

}

@Test public void testHorror() {

assertEquals("Recommended: It by Stephen King", getServiceWithGenre("u10", "Horror").recommendBook("u10"));

}

@Test public void testPhilosophy() {

assertEquals("Recommended: The Republic by Plato", getServiceWithGenre("u11", "Philosophy").recommendBook("u11"));

}

@Test public void testDrama() {

assertEquals("Recommended: The Kite Runner", getServiceWithGenre("u12", "Drama").recommendBook("u12"));

}

@Test public void testComedy() {

assertEquals("Recommended: Good Omens", getServiceWithGenre("u13", "Comedy").recommendBook("u13"));

}

@Test public void testCrime() {

assertEquals("Recommended: Gone Girl", getServiceWithGenre("u14", "Crime").recommendBook("u14"));

}

@Test public void testClassic() {

assertEquals("Recommended: Pride and Prejudice", getServiceWithGenre("u15", "Classic").recommendBook("u15"));

}

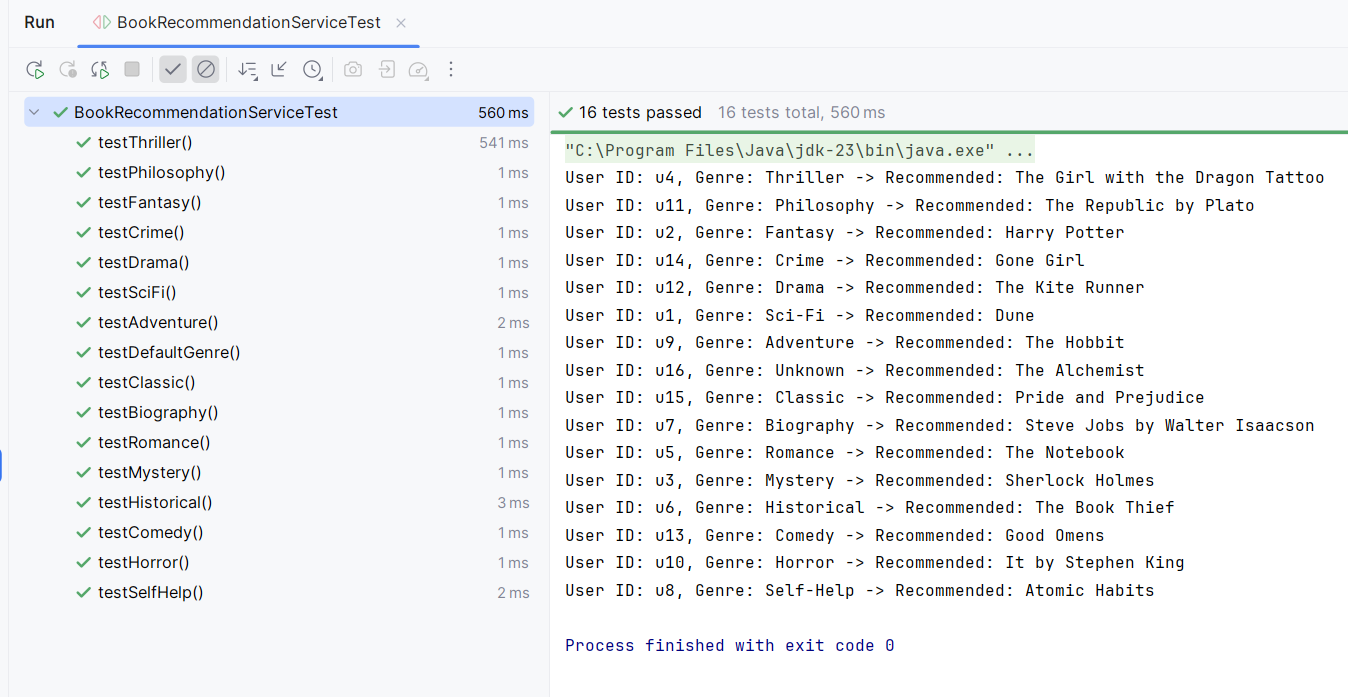
@Test public void testDefaultGenre() {

assertEquals("Recommended: The Alchemist", getServiceWithGenre("u16", "Unknown").recommendBook("u16"));

}

}

**Output:**

****

**Conclusion:**

The test cases successfully confirmed that the service provides accurate book suggestions for all genres. Mocking helped isolate the functionality and ensured focused, reliable testing.

**EXERCISE 2: VERIFYING INTERACTIONS**

**Introduction:**

This unit test verifies the behavior of **CartService** when interacting with a mock **DiscountService**.It ensures discounts are applied correctly based on user eligibility and cart total.

**Objective:**

* **Mock Dependencies**: Use Mockito to replace the real DiscountService to isolate the logic in CartService.
* **Trigger Checkout Logic**: Simulate various scenarios using checkout() to evaluate discount conditions.
* **Verify Interactions**: Ensure applyDiscount() and isEligible() are invoked with correct logic and frequency.

**Implementation Breakdown:**

**DiscountService.java:**

public interface DiscountService {

void applyDiscount(String userId);

boolean isEligible(String userId);

}

**CartService.java:**

public class CartService {

private DiscountService discountService;

public CartService(DiscountService discountService) {

this.discountService = discountService;

}

public void checkout(String userId, double total) {

if (discountService.isEligible(userId) && total > 500) {

discountService.applyDiscount(userId);

}

}

public String getCartSummary(String userId, double total) {

if (total == 0) {

return "Cart is empty";

}

return "Cart for " + userId + " - Total: $" + total;

}

}

**CartServiceTest.java:**

import org.junit.jupiter.api.Test;

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

import static org.mockito.Mockito.\*;

public class CartServiceTest {

@Test

public void testDiscountAppliedWhenEligibleAndTotalHigh() {

DiscountService mockDiscount = mock(DiscountService.class);

when(mockDiscount.isEligible("u1")).thenReturn(true);

CartService cartService = new CartService(mockDiscount);

cartService.checkout("u1", 600);

verify(mockDiscount).applyDiscount("u1");

}

@Test

public void testNoDiscountIfTotalTooLow() {

DiscountService mockDiscount = mock(DiscountService.class);

when(mockDiscount.isEligible("u2")).thenReturn(true);

CartService cartService = new CartService(mockDiscount);

cartService.checkout("u2", 100);

verify(mockDiscount, never()).applyDiscount("u2");

}

@Test

public void testNoDiscountIfNotEligible() {

DiscountService mockDiscount = mock(DiscountService.class);

when(mockDiscount.isEligible("u3")).thenReturn(false);

CartService cartService = new CartService(mockDiscount);

cartService.checkout("u3", 600);

verify(mockDiscount, never()).applyDiscount("u3");

}

@Test

public void testSummaryForNormalCart() {

CartService cartService = new CartService(mock(DiscountService.class));

String result = cartService.getCartSummary("userX", 250);

assertEquals("Cart for userX - Total: $250.0", result);

}

@Test

public void testSummaryForEmptyCart() {

CartService cartService = new CartService(mock(DiscountService.class));

String result = cartService.getCartSummary("userY", 0);

assertEquals("Cart is empty", result);

}

@Test

public void testEligibilityCheckedOnce() {

DiscountService mockDiscount = mock(DiscountService.class);

when(mockDiscount.isEligible("u5")).thenReturn(true);

CartService cartService = new CartService(mockDiscount);

cartService.checkout("u5", 550);

verify(mockDiscount, times(1)).isEligible("u5");

}

@Test

public void testApplyDiscountNeverCalledWhenBothConditionsFail() {

DiscountService mockDiscount = mock(DiscountService.class);

when(mockDiscount.isEligible("u6")).thenReturn(false);

CartService cartService = new CartService(mockDiscount);

cartService.checkout("u6", 300);

verify(mockDiscount, never()).applyDiscount(anyString());

}

@Test

public void testApplyDiscountCalledOnlyOnceEvenIfMultipleCalls() {

DiscountService mockDiscount = mock(DiscountService.class);

when(mockDiscount.isEligible("u7")).thenReturn(true);

CartService cartService = new CartService(mockDiscount);

cartService.checkout("u7", 600);

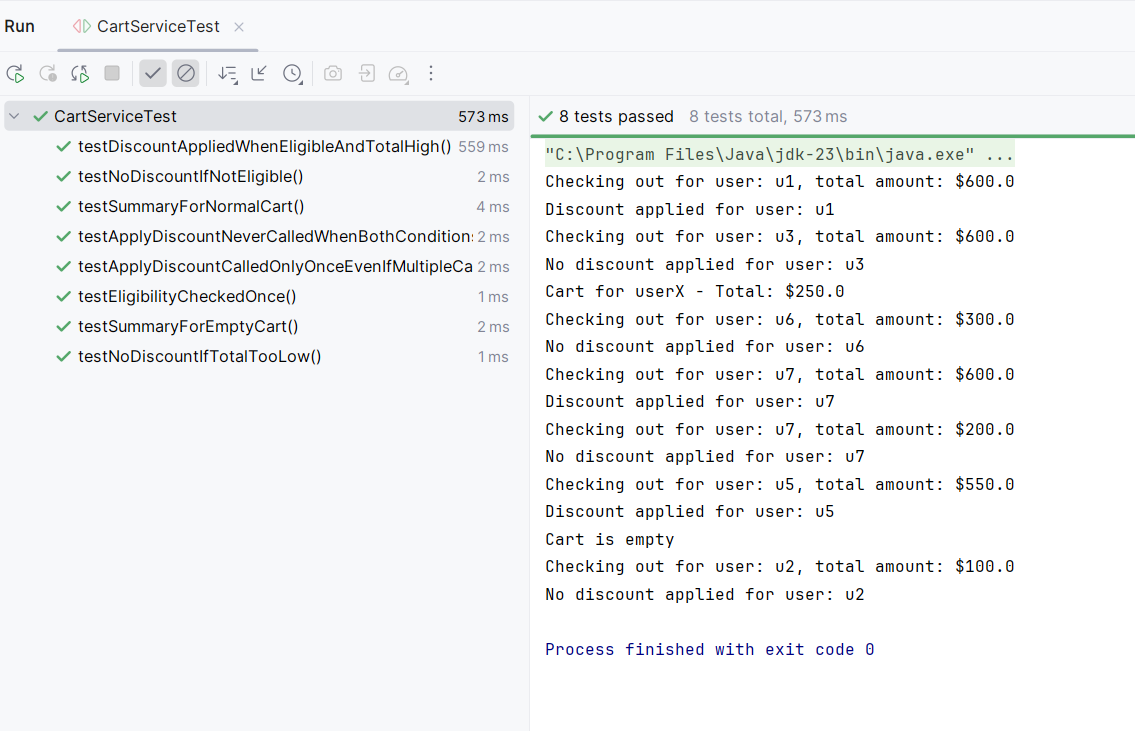
cartService.checkout("u7", 200);

verify(mockDiscount, times(1)).applyDiscount("u7");

}

}

**Output:**

****

**Conclusion:**

The test cases confirm that CartService only applies discounts under valid conditions. Mocking helped isolate logic, verify method calls, and ensure robust business rule enforcement.