**Exemple détaillé de code et de tests avec JUnit 5 et Mockito 4 en Java Maven**

**Scénario:**

Imaginons un scénario où vous avez une boutique en ligne qui vend des produits. Chaque produit a un nom, un prix et une quantité. Vous souhaitez créer un service qui calcule le prix total d'une commande en fonction des produits du panier.

**Code:**

// Classe Produit représentant un produit dans la boutique

import java.util.Objects; // Import pour la classe Objects

import java.util.List; // Import pour la classe List

class Produit {

private String nom;

private double prix;

private int quantité;

public Produit(String nom, double prix, int quantité) {

this.nom = nom;

this.prix = prix;

this.quantité = quantité;

}

public String getNom() {

return nom;

}

public double getPrix() {

return prix;

}

public int getQuantité() {

return quantité;

}

}

// Classe Commande représentant la commande d'un client

class Commande {

private List<Produit> produits;

public Commande(List<Produit> produits) {

this.produits = produits;

}

public double calculerPrixTotal() {

double prixTotal = 0.0;

for (Produit produit : produits) {

prixTotal += produit.getPrix() \* produit.getQuantité();

}

return prixTotal;

}

}

// Classe OrderService responsable du calcul du total des commandes

class OrderService {

private DiscountService serviceReduction;

public OrderService(DiscountService serviceReduction) {

this.serviceReduction = serviceReduction;

}

public double calculerTotalCommande(Commande commande) {

double prixTotal = commande.calculerPrixTotal();

double reduction = serviceReduction.appliquerReduction(prixTotal);

return prixTotal - reduction;

}

}

// Interface DiscountService définissant le comportement de calcul des réductions

interface DiscountService {

double appliquerReduction(double montant);

}

// Classe FixedDiscountService implémentant une stratégie de réduction fixe

class FixedDiscountService implements DiscountService {

private double montantReduction;

public FixedDiscountService(double montantReduction) {

this.montantReduction = montantReduction;

}

@Override

public double appliquerReduction(double montant) {

if (montant >= montantReduction) {

return montantReduction;

} else {

return montant;

}

}

}

**Tests avec JUnit 5 et Mockito:**

// Classe OrderServiceTest contenant les tests unitaires pour OrderService

import org.junit.jupiter.api.BeforeEach; // Import pour l'annotation @BeforeEach

import org.junit.jupiter.api.Test; // Import pour l'annotation @Test

import org.mockito.Mock; // Import pour l'annotation @Mock

import org.mockito.Mockito; // Import pour la classe Mockito

import org.mockito.junit.jupiter.MockitoExtension; // Import pour l'extension Mockito JUnit 5

import static org.junit.jupiter.api.Assertions.assertEquals; // Import pour l'assertion assertEquals

@ExtendWith(MockitoExtension.class)

class OrderServiceTest {

@Mock

private DiscountService serviceReduction;

@InjectMocks

private OrderService orderService;

@Test

void testCalculerTotalCommandeSansReduction() {

Produit produit1 = new Produit("Produit 1", 10.0, 2);

Produit produit2 = new Produit("Produit 2", 5.0, 1);

List<Produit> produits = Arrays.asList(produit1, produit2);

Commande commande = new Commande(produits);

// Mocker discountService pour retourner une réduction de 0

when(serviceReduction.appliquerReduction(anyDouble())).thenReturn(0.0);

double totalAttendu = 25.0;

double totalReel = orderService.calculerTotalCommande(commande);

assertEquals(totalAttendu, totalReel);

}

@Test

void testCalculerTotalCommandeAvecReductionFixe() {

Produit produit1 = new Produit("Produit 1", 20.0, 1);

Produit produit2 = new Produit("Produit 2", 15.0, 2);

List<Produit> produits = Arrays.asList(produit1, produit2);

Commande commande = new Commande(produits);

// Mocker discountService pour appliquer une réduction fixe de 10

when(serviceReduction.appliquerReduction(anyDouble())).thenReturn(10.0);

double totalAttendu = 45.0;

double totalReel = orderService.calculerTotalCommande(commande);

assertEquals(totalAttendu, totalReel);

}

}

**Explication:**

1. **Classe Produit:** Représente un produit avec un nom, un prix et une quantité.
2. **Classe Commande:** Représente la commande d'un client contenant une liste de produits

**Tests avec Mockito:**

Les tests de l'OrderService utilisent Mockito pour mocker le comportement du DiscountService. Cela permet de tester le OrderService de manière isolée sans dépendre d'une implémentation réelle du DiscountService.

* **@Mock**: L'annotation @Mock est utilisée pour créer une instance mockée du DiscountService. Cette instance n'a pas de comportement réel, mais elle peut être programmée pour répondre à des appels de méthode spécifiques.
* **@InjectMocks**: L'annotation @InjectMocks est utilisée pour injecter l'instance mockée du DiscountService dans le OrderService. Cela permet au OrderService d'utiliser le DiscountService mocké lors de l'exécution des tests.
* **when() et thenReturn()**: La méthode when() est utilisée pour spécifier un appel de méthode à mocker. La méthode thenReturn() est utilisée pour spécifier la valeur à retourner pour l'appel de méthode moqué.

Dans les tests, le DiscountService mocké est programmé pour retourner une réduction de 0,0 dans le premier test et une réduction fixe de 10,0 dans le second test. Cela permet de vérifier que le OrderService calcule correctement le total de la commande avec et sans réduction.

Cet exemple démontre l'utilisation des patterns de conception Strategy et Decorator dans le code Java, ainsi que l'utilisation de JUnit 5 et Mockito pour tester le code. Les tests unitaires garantissent que le code fonctionne comme prévu et qu'il est exempt de bogues.

**Points clés à retenir:**

* Le patron de conception Strategy permet d'encapsuler des algorithmes interchangeables, ce qui rend le code plus flexible et plus facile à maintenir.
* Le patron de conception Decorator permet d'étendre les fonctionnalités d'un objet sans modifier sa classe d'origine, ce qui favorise la composition et la réutilisabilité du code.
* JUnit 5 est un framework de test unitaire populaire pour Java qui offre une API claire et expressive pour écrire des tests unitaires.
* Mockito est un framework de mocking populaire pour Java qui permet de créer des objets mockés pour tester le code.

En combinant ces outils et techniques, les développeurs Java peuvent écrire du code propre, testable et maintenable.

L'exemple précédent peut être enrichi de plusieurs manières pour le rendre plus complet et informatif. Voici quelques idées :

**1. Assertions avancées:**

* **Utiliser des assertions spécifiques pour des types de données spécifiques:** Au lieu d'utiliser assertEquals pour tous les types de données, vous pouvez utiliser des assertions plus précises comme assertEqualsInts, assertEqualsDoubles, ou assertEqualsStrings.
* **Vérifier plusieurs valeurs avec une seule assertion:** Vous pouvez utiliser des méthodes comme assertAll pour regrouper plusieurs assertions et les vérifier en une seule fois.
* **Utiliser des messages d'échec personnalisés:** Vous pouvez fournir des messages d'échec personnalisés pour vos assertions afin de faciliter la compréhension des causes des échecs de tests.

**Exemple d'assertions avancées:**

@Test

void testCalculerTotalCommandeAvecReductionFixe() {

Produit produit1 = new Produit("Produit 1", 20.0, 1);

Produit produit2 = new Produit("Produit 2", 15.0, 2);

List<Produit> produits = Arrays.asList(produit1, produit2);

Commande commande = new Commande(produits);

// Mocker discountService pour appliquer une réduction fixe de 10

when(discountService.appliquerReduction(anyDouble())).thenReturn(10.0);

double totalAttendu = 35.0;

// Utiliser assertEqualsInts pour vérifier la valeur entière du total

assertEqualsInts(totalAttendu, orderService.calculerTotalCommande(commande), "Le total de la commande ne correspond pas à la valeur attendue");

// Vérifier plusieurs valeurs avec assertAll

assertAll(

() -> assertEqualsInts(totalAttendu, orderService.calculerTotalCommande(commande)),

() -> assertTrue(orderService.calculerTotalCommande(commande) > 0),

() -> assertFalse(orderService.calculerTotalCommande(commande) < 0)

);

}

**2. Tests d'échec:**

* **Tester les scénarios d'échec attendus:** Écrivez des tests qui vérifient que le code se comporte comme prévu dans les cas où des erreurs ou des conditions inattendues se produisent.
* **Utiliser des exceptions attendues:** Utilisez l'annotation @Test(expected = Exception.class) pour indiquer qu'un test doit lever une exception spécifique.

**Exemple de tests d'échec:**

@Test

void testCalculerTotalCommandeAvecProduitNull() {

List<Produit> produits = null;

Commande commande = new Commande(produits);

assertThrows(NullPointerException.class, () -> {

orderService.calculerTotalCommande(commande);

}, "Un NullPointerException doit être lancé liste de produit vide");

}

**3. Tests paramétrés:**

* **Tester plusieurs scénarios avec un ensemble de données:** Utilisez la fonctionnalité de tests paramétrés de JUnit 5 pour exécuter le même test avec différents ensembles de données.
* **Utiliser des sources de données personnalisées:** Vous pouvez créer vos propres sources de données pour fournir des données d'entrée aux tests paramétrés.

**Exemple de tests paramétrés:**

@ParameterizedTest(name = "Test calculerTotalCommande avec différents produits et réductions")

@ValueSource(doubles = {0.0, 5.0, 10.0})

@ValueSource(strings = {"Produit 1", "Produit 2", "Produit 3"})

void testCalculerTotalCommandeAvecParametres(double reduction, String nomProduit) {

// Créer des produits et une commande avec les paramètres fournis

Produit produit = new Produit(nomProduit, 10.0, 1);

List<Produit> produits = Arrays.asList(produit);

Commande commande = new Commande(produits);

// Mocker discountService pour appliquer la réduction spécifiée

when(discountService.appliquerReduction(anyDouble())).thenReturn(reduction);

// Calculer le total de la commande et vérifier le résultat

double totalAttendu = (10.0 - reduction);

double totalReel = orderService.calculerTotalCommande(commande);

assertEquals(totalAttendu, totalReel, 0.01);

}

**4. Matchers:**

**Utiliser des matchers pour vérifier des objets complexes:**

Les matchers JUnit 5 fournissent des moyens flexibles de vérifier l'état et le comportement des objets. Ils permettent de vérifier des propriétés spécifiques des objets, comme leur contenu, leur type, ou leur comportement lorsqu'ils sont invoqués.

**Exemple d'utilisation de matchers:**

@Test

void testCalculerTotalCommandeAvecProduitEtReduction() {

Produit produit = new Produit("Produit 1", 20.0, 2);

List<Produit> produits = Arrays.asList(produit);

Commande commande = new Commande(produits);

// Mocker discountService pour appliquer une réduction de 10

when(discountService.appliquerReduction(anyDouble())).thenReturn(10.0);

double totalAttendu = 35.0;

// Vérifier que la commande contient le produit attendu

assertThat(commande.getProduits(), contains(produit));

// Vérifier que le prix total de la commande correspond à la valeur attendue

assertThat(orderService.calculerTotalCommande(commande), is(equalTo(totalAttendu)));

}

**Combiner des matchers pour des assertions plus complexes:**

Les matchers peuvent être combinés pour créer des assertions plus complexes. Par exemple, vous pouvez utiliser and() pour combiner plusieurs conditions, ou or() pour vérifier si l'une ou plusieurs conditions sont vraies.

**Exemple de combinaison de matchers:**

@Test

void testCalculerTotalCommandeAvecProduitEtReductionEtPrixPositif() {

Produit produit = new Produit("Produit 1", 20.0, 2);

List<Produit> produits = Arrays.asList(produit);

Commande commande = new Commande(produits);

// Mocker discountService pour appliquer une réduction de 10

when(discountService.appliquerReduction(anyDouble())).thenReturn(10.0);

double totalAttendu = 35.0;

// Vérifier que la commande contient le produit attendu et que son prix est positif

assertThat(commande.getProduits(), contains(produit), everyItem(hasField("prix", greaterThan(0.0))));

// Vérifier que le prix total de la commande correspond à la valeur attendue et est positif

assertThat(orderService.calculerTotalCommande(commande), is(equalTo(totalAttendu)), is(greaterThan(0.0)));

}

L'utilisation d'assertions avancées, de tests d'échec, de tests paramétrés et de matchers permet d'écrire des tests unitaires plus complets, plus informatifs et plus robustes. Cela permet de garantir que le code fonctionne comme prévu et qu'il est exempt de bogues.

le fichier pom.xml complet correspondant aux exemples de code et de tests, avec les imports nécessaires :

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>order-service-example</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.openjdk.java</groupId>

<artifactId>java-base</artifactId>

<version>17</version>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>5.8.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.7.0</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>