1. **Crisp Placement Training**CourseDAO  
  
package com.dao;

import java.util.ArrayList;

import java.util.List;

import com.exception.InvalidCourseException;

import com.model.Course;

public class CourseDAO {

private final List<Course> courseList = new ArrayList<>();

public void addCourse(Course course) {

if (course != null) {

courseList.add(course);

}

}

public Course getCourseByCoordinator(String instructorName) throws InvalidCourseException {

if (courseList.isEmpty()) {

throw new InvalidCourseException("Course list is empty");

}

for (Course course : courseList) {

if (course.getCourseInstructor().equals(instructorName)) {

return course;

}

}

throw new InvalidCourseException("Invalid course instructor: " + instructorName);

}

}

EnrollmentDAO

package com.dao;

import java.util.ArrayList;

import java.util.List;

import com.exception.InvalidEnrollmentException;

import com.model.Enrollment;

public class EnrollmentDAO {

private final List<Enrollment> enrollmentList = new ArrayList<>();

public List<Enrollment> getEnrollmentList() {

return new ArrayList<>(enrollmentList); // Return a copy to preserve encapsulation

}

public void addEnrollment(Enrollment enrollment) throws InvalidEnrollmentException {

if (enrollment == null) {

throw new InvalidEnrollmentException("Enrollment cannot be null");

}

enrollmentList.add(enrollment);

}

public List<Enrollment> getEnrollmentsByCourseId(String courseId) {

List<Enrollment> matchingEnrollments = new ArrayList<>();

for (Enrollment enrollment : enrollmentList) {

if (enrollment.getCourse().getCourseId().equals(courseId)) {

matchingEnrollments.add(enrollment);

}

}

return matchingEnrollments;

}

}

**StudentDAO**

package com.dao;

import com.model.Student;

import com.exception.InvalidStudentException;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class StudentDAO {

private final List<Student> studentList = new ArrayList<>();

public void addStudent(Student student) {

if (student != null) {

studentList.add(student);

}

}

public boolean deleteStudent(String studentId) throws InvalidStudentException {

if (studentList.isEmpty()) {

throw new InvalidStudentException("No students in the list");

}

Iterator<Student> iterator = studentList.iterator();

while (iterator.hasNext()) {

Student student = iterator.next();

if (student.getStudentId().equals(studentId)) {

iterator.remove(); // Safe removal during iteration

return true;

}

}

throw new InvalidStudentException("Student ID does not exist: " + studentId);

}

public List<Student> getAllStudents() throws InvalidStudentException {

if (studentList.isEmpty()) {

throw new InvalidStudentException("No students found");

}

return new ArrayList<>(studentList); // Defensive copy

}

public Student getStudentById(String studentId) throws InvalidStudentException {

if (studentList.isEmpty()) {

throw new InvalidStudentException("Student list is empty");

}

for (Student student : studentList) {

if (student.getStudentId().equals(studentId)) {

return student;

}

}

throw new InvalidStudentException("Student ID not found: " + studentId);

}

}

**Course**

package com.model;

public class Course {

private String courseId;

private String courseName;

private String courseInstructor;

private double courseFee;

public Course() {

// Default constructor

}

public Course(String courseId, String courseName, String courseInstructor, double courseFee) {

this.courseId = courseId;

this.courseName = courseName;

this.courseInstructor = courseInstructor;

this.courseFee = courseFee;

}

public String getCourseId() {

return courseId;

}

public void setCourseId(String courseId) {

this.courseId = courseId;

}

public String getCourseName() {

return courseName;

}

public void setCourseName(String courseName) {

this.courseName = courseName;

}

public String getCourseInstructor() {

return courseInstructor;

}

public void setCourseInstructor(String courseInstructor) {

this.courseInstructor = courseInstructor;

}

public double getCourseFee() {

return courseFee;

}

public void setCourseFee(double courseFee) {

this.courseFee = courseFee;

}

}

**Enrollment**

package com.model;

import java.util.Date;

public class Enrollment {

private String enrollmentId;

private Student student;

private Course course;

private Date enrollmentDate;

private String enrollmentStatus;

private String group;

public Enrollment() {

// Default constructor

}

public Enrollment(String enrollmentId, Student student, Course course, Date enrollmentDate, String enrollmentStatus) {

this.enrollmentId = enrollmentId;

this.student = student;

this.course = course;

this.enrollmentDate = enrollmentDate;

this.enrollmentStatus = enrollmentStatus;

}

public String getEnrollmentId() {

return enrollmentId;

}

public void setEnrollmentId(String enrollmentId) {

this.enrollmentId = enrollmentId;

}

public Student getStudent() {

return student;

}

public void setStudent(Student student) {

this.student = student;

}

public Course getCourse() {

return course;

}

public void setCourse(Course course) {

this.course = course;

}

public Date getEnrollmentDate() {

return new Date(enrollmentDate.getTime()); // Defensive copy

}

public void setEnrollmentDate(Date enrollmentDate) {

if (enrollmentDate != null) {

this.enrollmentDate = new Date(enrollmentDate.getTime()); // Defensive copy

}

}

public String getEnrollmentStatus() {

return enrollmentStatus;

}

public void setEnrollmentStatus(String enrollmentStatus) {

this.enrollmentStatus = enrollmentStatus;

}

public String getGroup() {

return group;

}

public void setGroup(String group) {

this.group = group;

}

}

**Student**

package com.model;

public class Student {

private String studentId;

private String studentName;

private int age;

private long phoneNumber;

private String emailId;

public Student() {

// Default constructor

}

public Student(String studentId, String studentName, int age, long phoneNumber, String emailId) {

this.studentId = studentId;

this.studentName = studentName;

this.age = age;

this.phoneNumber = phoneNumber;

this.emailId = emailId;

}

public String getStudentId() {

return studentId;

}

public void setStudentId(String studentId) {

this.studentId = studentId;

}

public String getStudentName() {

return studentName;

}

public void setStudentName(String studentName) {

this.studentName = studentName;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public long getPhoneNumber() {

return phoneNumber;

}

public void setPhoneNumber(long phoneNumber) {

this.phoneNumber = phoneNumber;

}

public String getEmailId() {

return emailId;

}

public void setEmailId(String emailId) {

this.emailId = emailId;

}

}

2. **EB Connection  
  
EBBillTest**

package com.test;

import static org.junit.jupiter.api.Assertions.assertFalse;

import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Test;

import com.model.EBConnection;

import com.util.EBBill;

public class EBBillTest {

//Write the code for testing assertion using JUNIT

private EBConnection connection;

@BeforeEach

public void setUp() {

connection = new EBConnection("EB124567", "Domestic", 0);

}

//Add required annotation

//Test the calculateBillAmount method in EBBill class to check bill for the units is valid - Use assertTrue

@Test

public void test1\_calculateBillAmountForUnitLessThan100() {

double bill = EBBill.calculateBillAmount(80);

assertTrue(bill == 208.0, "Bill for 80 units should be 208.0");

}

//Add required annotation

//Test the calculateBillAmount method in EBBill class to check bill for the units is valid - Use assertTrue

@Test

public void test2\_calculateBillAmountForUnitEqualTo100() {

double bill = EBBill.calculateBillAmount(100);

assertTrue(bill == 325.0, "Bill for 100 units should be 325.0");

}

//Add required annotation

//Test the calculateBillAmount method in EBBill class to check bill for the units is valid - Use assertTrue

@Test

public void test3\_calculateBillAmountForUnitGreaterThan100() {

double bill = EBBill.calculateBillAmount(150);

assertTrue(bill == 487.5, "Bill for 150 units should be 487.5");

}

//Add required annotation

//Test the calculateBillAmount method in EBBill class to check bill for the units is valid - Use assertTrue

@Test

public void test4\_calculateBillAmountForUnitLessThan900() {

double bill = EBBill.calculateBillAmount(899);

assertTrue(bill == 2921.75, "Bill for 899 units should be 2921.75");

}

//Add required annotation

//Test the calculateBillAmount method in EBBill class to check bill for the units is valid - Use assertTrue

@Test

public void test5\_calculateBillAmountForUnitEqualTo900() {

double bill = EBBill.calculateBillAmount(900);

assertTrue(bill == 4734.0, "Bill for 900 units should be 4734.0");

}

//Add required annotation

//Test the calculateBillAmount method in EBBill class to check bill for the units is valid - Use assertTrue

@Test

public void test6\_calculateBillAmountForUnitGreaterThan900() {

double bill = EBBill.calculateBillAmount(950);

assertTrue(bill == 4997.0, "Bill for 950 units should be 4997.0");

}

//Add required annotation

//Test the validateConnectionType method in EBBill class when the connectionType is valid - Use assertTrue

@Test

public void test7\_validateConnectionType\_Industrial() {

boolean isValid = EBBill.validateConnectionType("Industrial");

assertTrue(isValid, "'Industrial' should be a valid connection type");

}

//Add required annotation

//Test the validateConnectionType method in EBBill class when the connectionType is valid - Use assertTrue

@Test

public void test8\_validateConnectionType\_Domestic() {

boolean isValid = EBBill.validateConnectionType("Domestic");

assertTrue(isValid, "'Domestic' should be a valid connection type");

}

//Add required annotation

//Test the validateConnectionType method in EBBill class when the connectionType is invalid - Use assertFalse

@Test

public void test9\_validateConnectionType\_Invalid() {

boolean isValid = EBBill.validateConnectionType("Commercial");

assertFalse(isValid, "'Commercial' should be an invalid connection type");

}

//Add required annotation

//Test the calculateBillAmount method in EBBill class to check bill for Invalid units - Use assertTrue

@Test

public void test10\_calculateBillAmountForAnInvalidUnits() {

double billAmount = EBBill.calculateBillAmount(-45);

assertTrue(billAmount >=0);

}

}

3. **CarcoExpress**ShipmentUtilTest

package com.test;

import static org.junit.jupiter.api.Assertions.assertFalse;

import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Test;

import com.model.Shipment;

import com.util.ShipmentUtil;

public class ShipmentUtilTest {

//Write the code for testing assertion using JUNIT

private static Shipment standardShipment;

private static Shipment expressShipment;

private static Shipment overnightShipment;

private static Shipment invalidShipment;

@BeforeAll

public static void setUp() {

standardShipment = new Shipment("ST0010", "Standard", 10.0);

expressShipment = new Shipment("ST0011", "Express", 8.0);

overnightShipment = new Shipment("ST0012", "Overnight", 24.0);

invalidShipment = new Shipment("ST0013", "Economy", 15.0); // invalid type

}

//Add required annotation

//Test the validateShipmentType method in ShipmentUtil class when the shipmentType is Standard - Use assertTrue

@Test

public void test1\_ValidateForValidShipmentTypeStandard() {

assertTrue(ShipmentUtil.validateShipmentType(standardShipment.getShipmentType()));

}

//Add required annotation

//Test the validateShipmentType method in ShipmentUtil class when the shipmentType is Express - Use assertTrue

@Test

public void test2\_ValidateForValidShipmentTypeExpress() {

assertTrue(ShipmentUtil.validateShipmentType(expressShipment.getShipmentType()));

}

//Add required annotation

//Test the validateShipmentType method in ShipmentUtil class when the shipmentType is Overnight - Use assertTrue

@Test

public void test3\_ValidateForValidShipmentTypeOvernight() {

assertTrue(ShipmentUtil.validateShipmentType(overnightShipment.getShipmentType()));

}

//Add required annotation

//Test the validateShipmentType method in ShipmentUtil class when the shipmentType is invalid - Use assertFalse

@Test

public void test4\_ValidateForInvalidShipmentType() {

assertFalse(ShipmentUtil.validateShipmentType(invalidShipment.getShipmentType()));

}

//Add required annotation

//Test the calculateShippingCost method in ShipmentUtil class when the shipmentType is Standard - Use assertTrue

@Test

public void test5\_CalculateShippingCostForStandard() {

double expected = 10.0 \* 50;

double actual = ShipmentUtil.calculateShippingCost("Standard", 10.0);

assertTrue(expected == actual, "Shipping cost for Standard should be " + expected);

}

//Add required annotation

//Test the calculateShippingCost method in ShipmentUtil class when the shipmentType is Express - Use assertTrue

@Test

public void test6\_CalculateShippingCostForExpress() {

double expected = 8.0 \* 100;

double actual = ShipmentUtil.calculateShippingCost("Express", 8.0);

assertTrue(expected == actual, "Shipping cost for Express should be " + expected);

}

//Add required annotation

//Test the calculateShippingCost method in ShipmentUtil class when the shipmentType is Overnight - Use assertTrue

@Test

public void test7\_CalculateShippingCostForOvernight() {

double expected = 24.0 \* 200;

double actual = ShipmentUtil.calculateShippingCost("Overnight", 24.0);

assertTrue(expected == actual, "Shipping cost for Overnight should be " + expected);

}

@Test

public void test8\_CalculateShippingCostForInvalidShipmentType() {

double actual = ShipmentUtil.calculateShippingCost("Economy", 15.0);

assertTrue(actual == 0.0, "Shipping cost for invalid type should be 0.0");

}

}

4. **Employee Details**

**EmployeeTest**package com.test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertFalse;

import static org.junit.jupiter.api.Assertions.assertNotNull;

import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Test;

import com.model.Employee;

import com.util.EmployeeUtil;

public class EmployeeTest

{

//Write the code for testing assertion using JUNIT

private static Employee employee;

//Add required annotation

@BeforeAll

public static void setUp() throws Exception

{

employee = new Employee("EMP/123/2345", "Asha", "FullTime", "Analyst", "asha@xyz.com");

}

//Add required annotation

//Test the validateEmployeeId method in Employee class when the employeeId is valid - Use assertTrue

@Test

public void test10\_checkForValidEmployeeId() {

boolean result = EmployeeUtil.validateEmployeeId(employee.getEmployeeId());

assertTrue(result, "Expected valid employee ID to return true");

}

//Add required annotation

//Test the validateEmployeeId method in Employee class when the employeeId is invalid - Use assertFalse

@Test

public void test11\_checkForInvalidEmployeeId() {

boolean result = EmployeeUtil.validateEmployeeId("emp123-456");

assertFalse(result, "Expected invalid employee ID to return false");

}

//Add required annotation

//Test the validateJobType method in Employee class when the jobType is valid ("FullTime") - Use assertEquals

@Test

public void test12\_checkForValidJobType() {

String result = EmployeeUtil.validateJobType(employee.getJobType());

assertEquals("FullTime", result, "Expected jobType to return 'FullTime'");

}

//Add required annotation

//Test the findEmployeeName method in Employee class when name is valid - Use assertEquals

@Test

public void test13\_checkForValidEmployeeName() {

String result = EmployeeUtil.findEmployeeName("Asha");

assertEquals("Asha", result, "Expected employeeName to return 'Asha'");

}

}

5. **Employee Appraisal**EmployeeAppraisalParameterizedTest

package com.test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.Arrays;

import java.util.stream.Stream;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.Arguments;

import org.junit.jupiter.params.provider.MethodSource;

import com.util.EmployeeAppraisalService;

public class EmployeeAppraisalParameterizedTest {

// Provide necessary annotations

@ParameterizedTest

@MethodSource("data")

public void testCalculateIncrement(String empId, String name, String dept, double salary, double rating,

double incrementedSalary) throws Exception {

double actual = EmployeeAppraisalService.calculateIncrement(salary, rating);

assertEquals(incrementedSalary, actual, 0.01, "Failed for employee ID: " + empId + ", rating: " + rating);

}

// Include necessary stream to provide data to the test method

static Stream<Arguments> data() {

return Stream.of(

Arguments.of("EMP001", "Alice", "HR", 50000.0, 1.0, 55000.0),

Arguments.of("EMP002", "Bob", "Finance", 60000.0, 3.0, 66000.0),

Arguments.of("EMP003", "Cleo", "IT", 70000.0, 3.5, 84000.0),

Arguments.of("EMP004", "Dylan", "Sales", 80000.0, 4.5, 104000.0),

Arguments.of("EMP005", "Ella", "Admin", 90000.0, 5.5, 90000.0) // invalid rating

);

}

}

6. **Employee Address Update**

EmployeeDAOTest

package com.test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertThrows;

import static org.junit.jupiter.api.Assertions.assertTrue;

import static org.mockito.ArgumentMatchers.anyString;

import static org.mockito.Mockito.\*;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import javax.sql.DataSource;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.junit.jupiter.MockitoExtension;

import com.util.EmployeeDAO;

import com.exception.AddressUpdateException;

@ExtendWith(MockitoExtension.class)

public class EmployeeDAOTest {

// Use appropriate annotation and create the necessary mock objects.

// Use appropriate annotation and inject the mock objects

@Mock

private DataSource dataSource;

@Mock

private Connection connection;

@Mock

private PreparedStatement preparedStatement;

@InjectMocks

private EmployeeDAO employeeDAO;

// Include appropriate annotation

@BeforeEach

public void setUp() throws SQLException {

lenient().when(dataSource.getConnection()).thenReturn(connection);

lenient().when(connection.prepareStatement(anyString())).thenReturn(preparedStatement);

lenient().when(preparedStatement.executeUpdate()).thenReturn(1);

}

// Include appropriate annotation

@Test

public void testInvalidEmployeeId() {

AddressUpdateException exception = assertThrows(AddressUpdateException.class, () ->

employeeDAO.addressChange("123", "New Address")

);

assertEquals("Invalid Employee Id", exception.getMessage());

}

// Include appropriate annotation

@Test

public void testInvalidEmployeeAddress() {

AddressUpdateException exception = assertThrows(AddressUpdateException.class, () ->

employeeDAO.addressChange("12345", null)

);

assertEquals("Invalid Employee Address", exception.getMessage());

}

// Include appropriate annotation

@Test

public void testInvalidEmployeeIdWithNull() {

AddressUpdateException exception = assertThrows(AddressUpdateException.class, () ->

employeeDAO.addressChange(null, "New Address")

);

assertEquals("Invalid Employee Id", exception.getMessage());

}

@Test

public void testMethodCall() throws SQLException, AddressUpdateException {

String empId = "12345";

String empAddress = "New Address";

boolean result = employeeDAO.addressChange(empId, empAddress);

assertTrue(result);

verify(dataSource, times(1)).getConnection();

verify(connection, times(1)).prepareStatement(anyString());

verify(preparedStatement, times(1)).executeUpdate();

}

}

7. **Computation Service**

MathApplicationTest

package com.test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertThrows;

import static org.junit.jupiter.api.Assertions.assertTrue;

import static org.mockito.Mockito.never;

import static org.mockito.Mockito.times;

import static org.mockito.Mockito.verify;

import static org.mockito.Mockito.when;

import com.util.Square;

import com.util.MathApplication;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)

public class MathApplicationTest {

// Use appropriate annotation and mock the required object

// Use appropriate annotation and inject the mocked object

@Mock

private Square square;

@InjectMocks

private MathApplication mathApplication;

// Include appropriate annotation

@Test

public void testSuccess() {

int validSide = 5;

mathApplication.assignSide(validSide);

// Verifies setSide is called exactly once

verify(square, times(1)).setSide(validSide);

}

// Include appropriate annotation

@Test

public void testFailed() {

int invalidSide = 0;

try {

mathApplication.assignSide(invalidSide);

} catch (RuntimeException e) {

// Expected, do nothing

}

// Verifies setSide is NEVER called when input is invalid

verify(square, never()).setSide(invalidSide);

}

// Include appropriate annotation

@Test

public void testException() {

RuntimeException exception = assertThrows(RuntimeException.class, () -> {

mathApplication.assignSide(-1);

});

assertEquals("Invalid Input", exception.getMessage());

}

}

8. **Employee Details (Mockito)**EmployeeTest

package com.test;

import java.util.ArrayList;

import java.util.List;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.mockito.Mock;

import org.mockito.InjectMocks;

import org.mockito.MockitoAnnotations;

import org.mockito.junit.jupiter.MockitoExtension;

import org.junit.jupiter.api.extension.ExtendWith;

import static org.mockito.Mockito.times;

import static org.mockito.Mockito.verify;

import static org.mockito.Mockito.when;

import static org.mockito.Mockito.doNothing;

import static org.junit.jupiter.api.Assertions.assertThrows;

import com.model.Employee;

import com.service.EmployeeService;

import com.repo.EmployeeRepo;

@ExtendWith(MockitoExtension.class)

public class EmployeeTest {

//Add the appropriate annotation

@Mock

EmployeeRepo repo;

//Add the appropriate annotation

@InjectMocks

EmployeeService service;

//Test the addEmployee method in EmployeeService class

@Test

public void test1AddEmployee() {

Employee e = new Employee(101, "John", "john@example.com", 50000);

when(repo.addEmployeeToList(e)).thenReturn(1);

int result = service.addEmployee(e);

assertEquals(1, result);

}

//Test the deleteEmployee method in EmployeeService class

@Test

public void test2DeleteEmployee() {

Employee e = new Employee(102, "Jane", "jane@example.com", 55000);

service.deleteEmployee(e);

verify(repo, times(1)).deleteEmployeeFromList(e);

}

//Test the fetchEmployeeById method in EmployeeService class for a valid employeeId

@Test

public void test3FetchEmployeeByEmployeeId() {

Employee e = new Employee(103, "Alice", "alice@example.com", 60000);

when(repo.getEmployeeByEmployeeId(103)).thenReturn(e);

Employee result = service.fetchEmployeeById(103);

assertEquals(e, result);

}

//Test the fetchEmployeeById method in EmployeeService class for an invalid employeeId

@Test

public void test4FetchEmployeeByEmployeeIdWhenNull() {

when(repo.getEmployeeByEmployeeId(104)).thenReturn(null);

assertThrows(NullPointerException.class, () -> service.fetchEmployeeById(104));

}

//Test the fetchEmployee method in EmployeeService class

@Test

public void test5FetchEmployee() {

List<Employee> list = new ArrayList<>();

list.add(new Employee(105, "Bob", "bob@example.com", 62000));

list.add(new Employee(106, "Eve", "eve@example.com", 64000));

when(repo.getEmployee()).thenReturn(list);

List<Employee> result = service.fetchEmployee();

assertEquals(2, result.size());

assertEquals("Bob", result.get(0).getEmployeeName());

}

}