Coding Task 7

Unit Testing

7. Create Unit test cases for Finance System are essential to ensure the correctness and reliability of your system. Following questions to guide the creation of Unit test cases:

7.1 Write test case to test if user created successfully

7.2 Write test case to test if expene is created successfully.

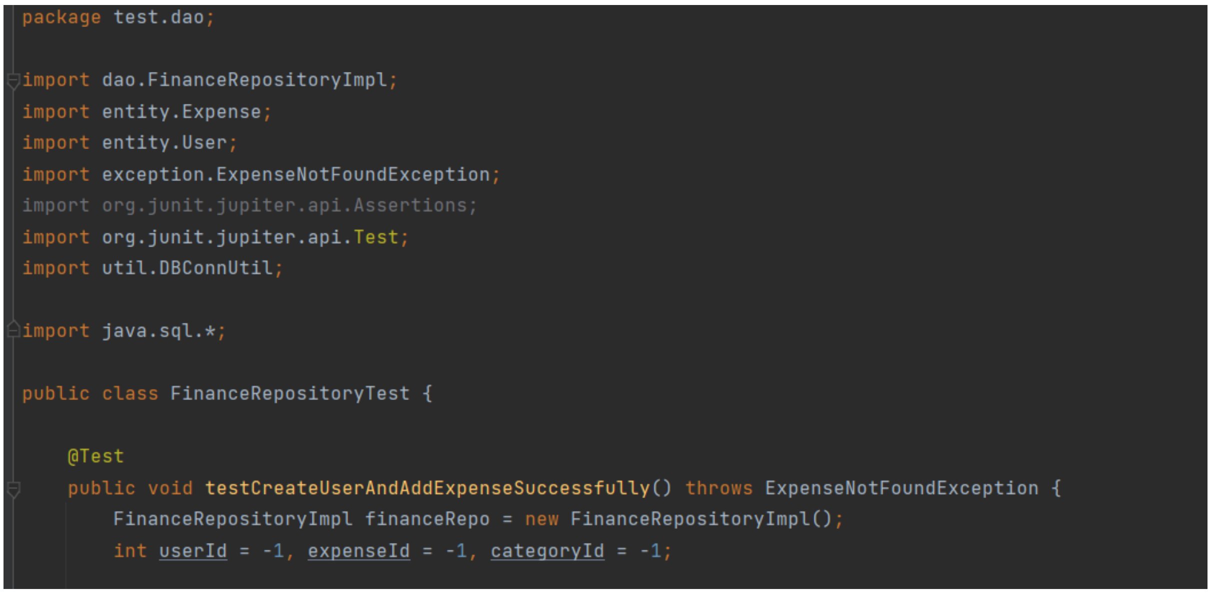
7.3 Write test case to test search of expense

7.4 Write test case to test if the exceptions are thrown correctly based on scenario

Unit testing is a crucial phase in software development that helps verify the functionality of individual components in isolation. In this Finance System, JUnit-based unit tests ensure the reliability, correctness, and robustness of the core functionalities like user creation, expense addition, data retrieval, and exception handling. Through well-structured test cases, we validate whether the system performs as expected under various scenarios.

Task 7.1 User Creation

This test validates if a new user is successfully created in the system. A test user is initialized with dummy data and passed to the createUser() method of the FinanceRepositoryImpl class. If the method returns true and a valid user ID is generated, the test is considered successful. This ensures the users table in the database handles insertions correctly and assigns unique identifiers.



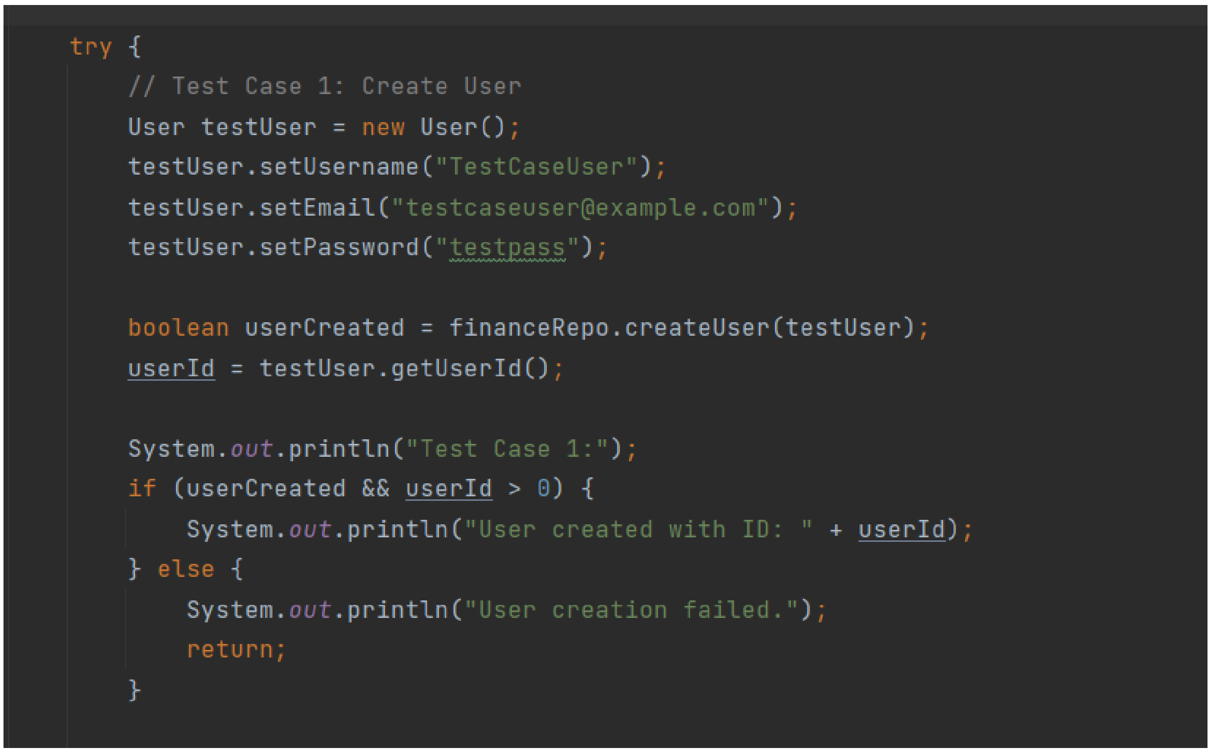
```java
package test.dao;

import dao.FinanceRepositoryImpl;
import entity.Expense;
import entity.User;
import exception.ExpenseNotFoundException;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import util.DBConnUtil;

import java.sql.*;

public class FinanceRepositoryTest {

    @Test
    public void testCreateUserAndAddExpenseSuccessfully() throws ExpenseNotFoundException {
        FinanceRepositoryImpl financeRepo = new FinanceRepositoryImpl();
        int userId = -1, expenseId = -1, categoryId = -1;
```



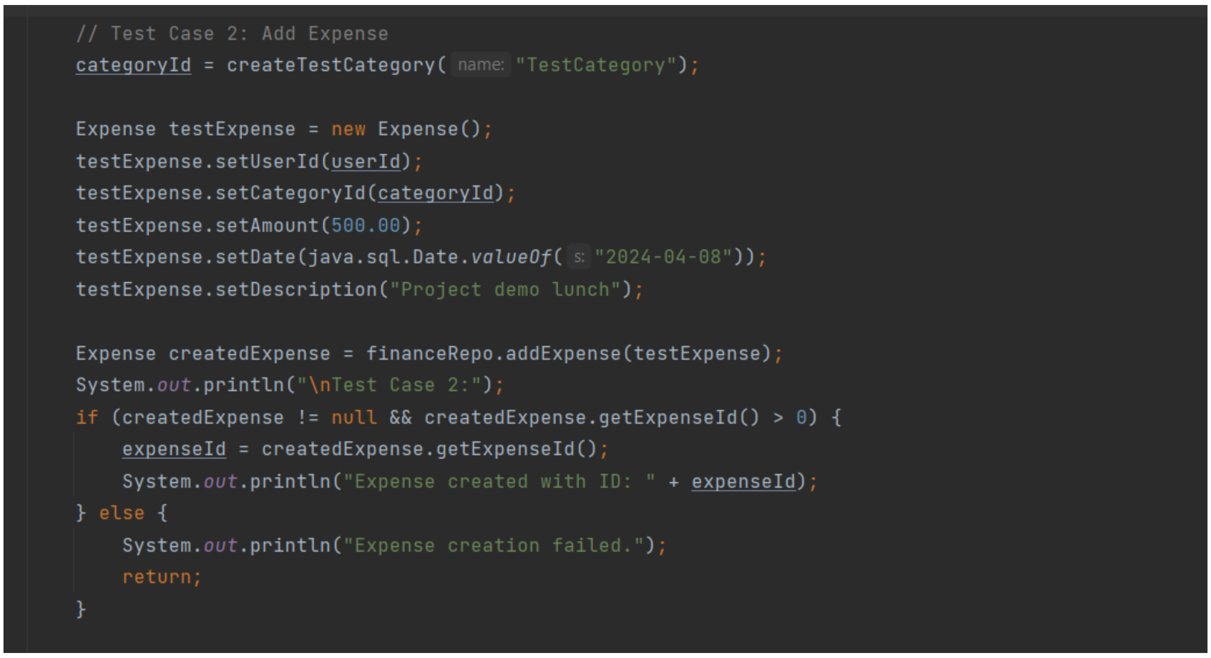
```java
try {
    // Test Case 1: Create User
    User testUser = new User();
    testUser.setUsername("TestCaseUser");
    testUser.setEmail("testcaseuser@example.com");
    testUser.setPassword("testpass");

    boolean userCreated = financeRepo.createUser(testUser);
    userId = testUser.getUserId();

    System.out.println("Test Case 1:");
    if (userCreated && userId > 0) {
        System.out.println("User created with ID: " + userId);
    } else {
        System.out.println("User creation failed.");
        return;
    }
```

Task 7.2 Expense Creation

In this case, the test confirms whether an expense is properly added for a given user. A test category is first created, and an expense object is initialized with all required details such as amount, date, description, and associated user and category IDs. After calling the addExpense() method, the system checks if a valid expense ID is generated. This verifies that the expense creation logic functions correctly and is integrated with user and category data.



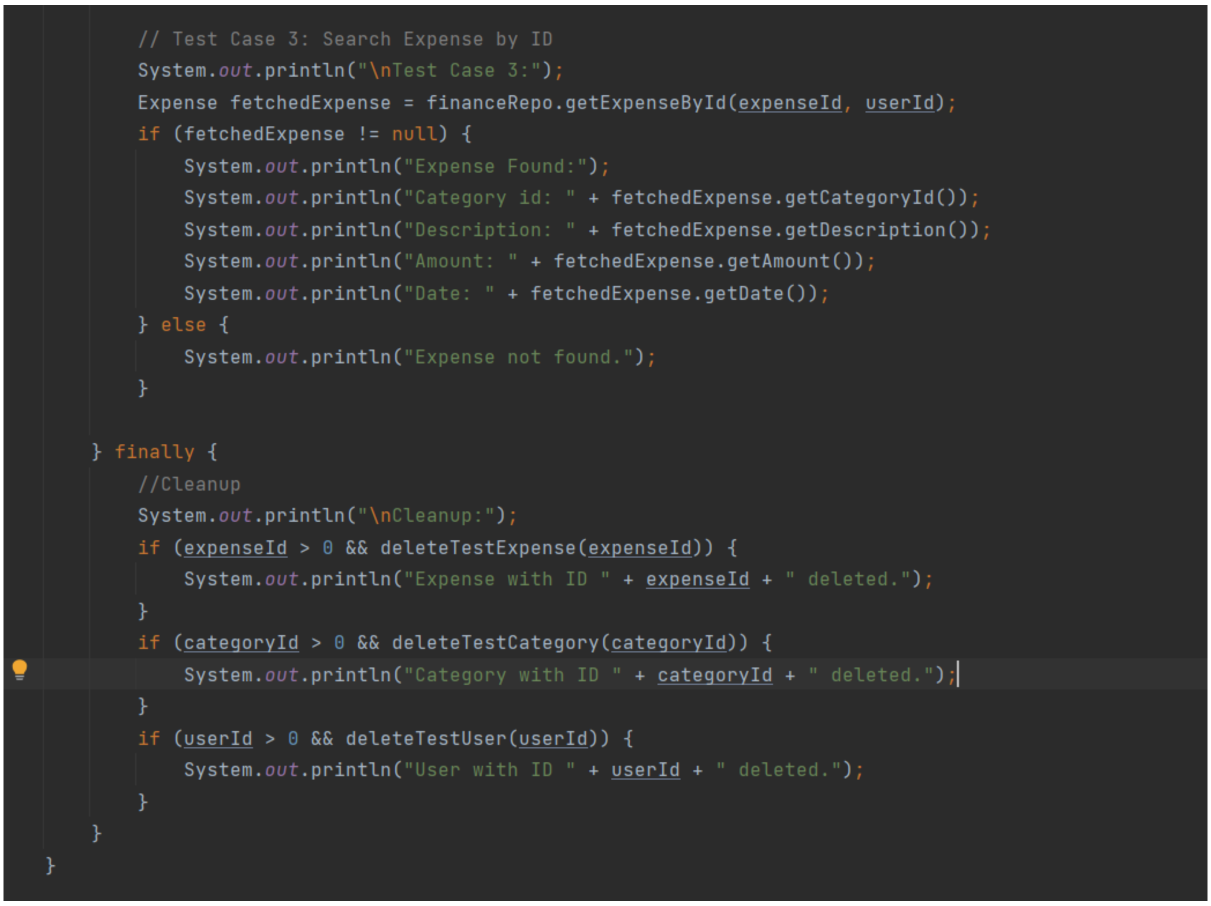
```java
// Test Case 2: Add Expense
categoryId = createTestCategory( name: "TestCategory");

Expense testExpense = new Expense();
testExpense.setUserId(userId);
testExpense.setCategoryId(categoryId);
testExpense.setAmount(500.00);
testExpense.setDate(java.sql.Date.valueOf( s: "2024-04-08"));
testExpense.setDescription("Project demo lunch");

Expense createdExpense = financeRepo.addExpense(testExpense);
System.out.println("\nTest Case 2:");
if (createdExpense != null && createdExpense.getExpenseId() > 0) {
    expenseId = createdExpense.getExpenseId();
    System.out.println("Expense created with ID: " + expenseId);
} else {
    System.out.println("Expense creation failed.");
    return;
}
```

Task 7.3 Search Expense by ID

This task verifies whether an added expense can be correctly retrieved using its ID. After the expense is created, the getExpenseById() method is used to fetch it from the database. The returned object is validated for its content (category ID, description, amount, and date). This test ensures that the retrieval process is accurate and reflects the latest state of the database for valid user and expense combinations.

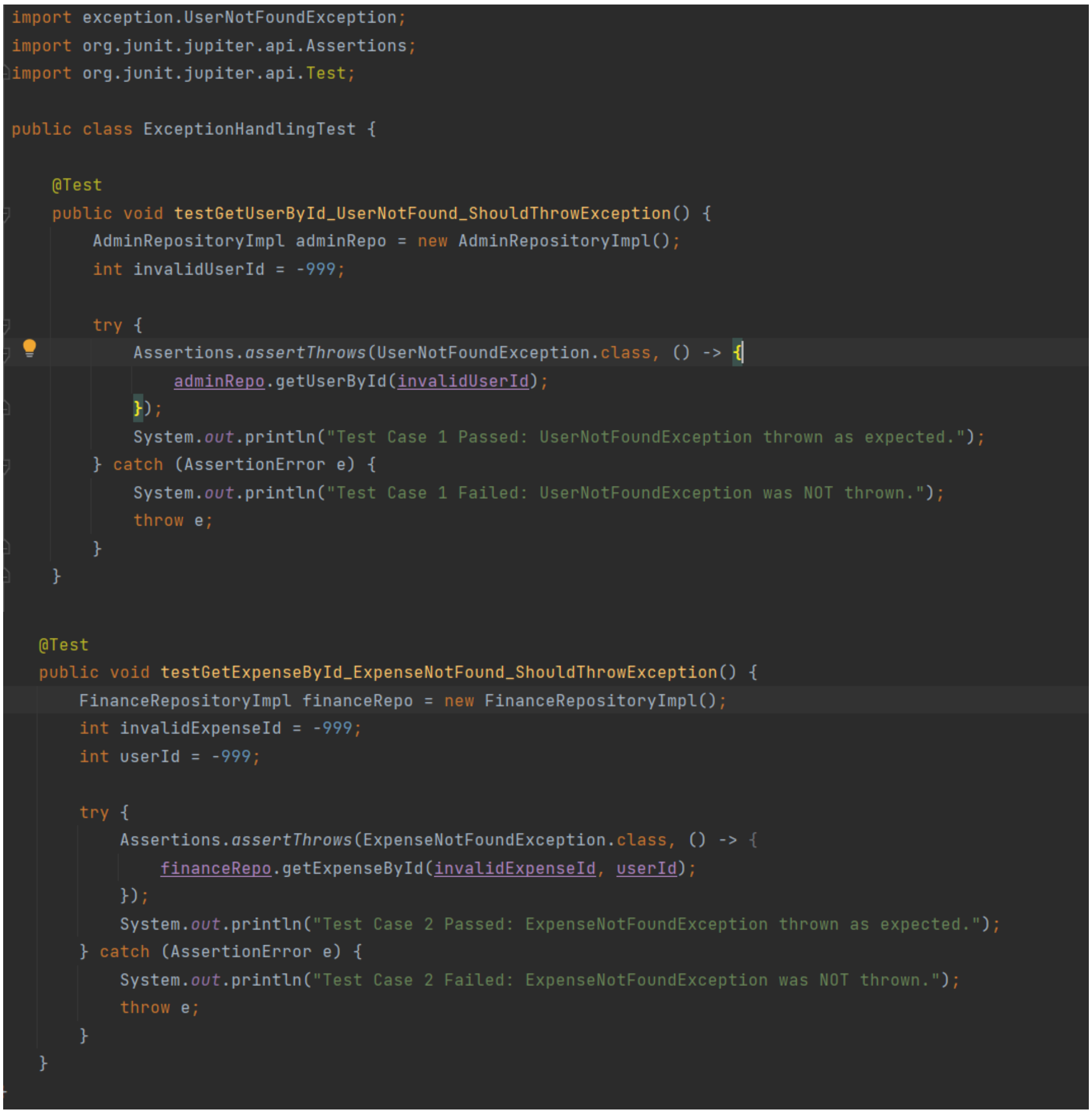


```java
            // Test Case 3: Search Expense by ID
            System.out.println("\nTest Case 3:");
            Expense fetchedExpense = financeRepo.getExpenseById(expenseId, userId);
            if (fetchedExpense != null) {
                System.out.println("Expense Found:");
                System.out.println("Category id: " + fetchedExpense.getCategoryId());
                System.out.println("Description: " + fetchedExpense.getDescription());
                System.out.println("Amount: " + fetchedExpense.getAmount());
                System.out.println("Date: " + fetchedExpense.getDate());
            } else {
                System.out.println("Expense not found.");
            }

        } finally {
            //Cleanup
            System.out.println("\nCleanup:");
            if (expenseId > 0 && deleteTestExpense(expenseId)) {
                System.out.println("Expense with ID " + expenseId + " deleted.");
            }
            if (categoryId > 0 && deleteTestCategory(categoryId)) {
                System.out.println("Category with ID " + categoryId + " deleted.");
            }
            if (userId > 0 && deleteTestUser(userId)) {
                System.out.println("User with ID " + userId + " deleted.");
            }
        }
    }
```

Task 7. 4 Exception Handling

1. **User Not Found :** This unit test checks the system's behavior when an invalid user ID is provided to the getUserById() method. The test asserts that a UserNotFoundException is thrown as expected. This ensures the application gracefully handles scenarios where data is not present, and the correct exception is thrown to inform the user or developer of the issue.
2. **Expense Not Found :** Similar to the user exception case, this test validates whether the getExpenseById() method throws an ExpenseNotFoundException when an invalid expense ID and user ID are passed. By confirming the exception behavior, this test ensures robustness and prevents unexpected crashes when dealing with nonexistent data.



```java
import exception.UserNotFoundException;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class ExceptionHandlingTest {

    @Test
    public void testGetUserById_UserNotFound_ShouldThrowException() {
        AdminRepositoryImpl adminRepo = new AdminRepositoryImpl();
        int invalidUserId = -999;

        try {
            Assertions.assertThrows(UserNotFoundException.class, () -> {
                adminRepo.getUserById(invalidUserId);
            });
            System.out.println("Test Case 1 Passed: UserNotFoundException thrown as expected.");
        } catch (AssertionError e) {
            System.out.println("Test Case 1 Failed: UserNotFoundException was NOT thrown.");
            throw e;
        }
    }

  @Test
  public void testGetExpenseById_ExpenseNotFound_ShouldThrowException() {
      FinanceRepositoryImpl financeRepo = new FinanceRepositoryImpl();
      int invalidExpenseId = -999;
      int userId = -999;

      try {
          Assertions.assertThrows(ExpenseNotFoundException.class, () -> {
              financeRepo.getExpenseById(invalidExpenseId, userId);
          });
          System.out.println("Test Case 2 Passed: ExpenseNotFoundException thrown as expected.");
      } catch (AssertionError e) {
          System.out.println("Test Case 2 Failed: ExpenseNotFoundException was NOT thrown.");
          throw e;
      }
  }
```

Conclusion:

Through these unit tests, the core functionalities of the Finance System such as user and expense creation, data fetching, and error handling are validated for correctness and reliability. The tests not only verify successful operations but also ensure that failures are managed gracefully with appropriate exception handling. This structured testing approach improves software quality, makes debugging easier, and enhances overall system robustness before deploying to production.
