Seminar 3

Object-Oriented Design, IV1350

Dana Ghafour, danagf@kth.se

2023-05-04

Contents

1 Introduction

I worked with Ali Madhool Madhool when solving these tasks. We collaborated through online meetings and shared our code and test cases on Discord. We divided the work equally and checked each other's code and tests for errors and improvements.

Task 1

A Java program was written that implements the basic flow, the alternative flow 3-4b, and the startup scenario for processing a sale in a point-of-sale system. We had to use the class diagram that we designed in assignment two as a guide for creating the classes, attributes, methods, and associations. We also had to follow the requirements specification given in assignment one and ensure that our code is compilable, executable, well-commented, and follows the guidelines in chapter six of the textbook.

Task 2

For this task, we had to write unit tests for our program using JUnit. We had to write tests for all classes in the controller, model, and integration layers, except for those that only have getters and constructors. We also had to ignore testing methods that only produce output to System.out. We had to write new test classes and not use the given tests from the textbook or *practice programming.*

2 Method

Task 1

As our methodology, we used object-oriented design principles and data transfer objects (DTOs) to write a Java program for a point-of-sale system. The implementation of the requirements specification and the design we created for the earlier seminar served as the foundation for our program. We believed the program would be more modular, reusable, and maintainable if classes and methods were used to represent and manage various entities and concepts in the system. We identified the main entities and concepts in the system, such as Controller, AccountingSystem, InventorySystem, ReceiptPrinter, Payment, Product, Receipt, and Sale, and created classes for each of them. We also defined the attributes and methods for each class to encapsulate their state and behavior. DTOs were used to transfer information between classes, by which we could conceal the inner details of various classes. To simulate the interaction between the customer and the cashier, we used hard-coded method calls. We also made sure that the program updated the external systems and printed a receipt during a sale.

Task 2

For this task, we wrote unit tests for different classes in the controller, model, and integration layers of the process sale program. For each method and class, we created a test class with the same name followed by Test. We used the JUnit 5 as the testing framework and imported the necessary libraries. We used the @BeforeEach annotation to initialize the objects needed for testing before each test method, and the @Test annotation to mark each test method. We used the assertEquals method to compare the actual and expected values of the methods under test. We also used assertDoesNotThrow and assertNotNull methods to check that some methods did not throw exceptions or returned null values.

3 Result

Task 1

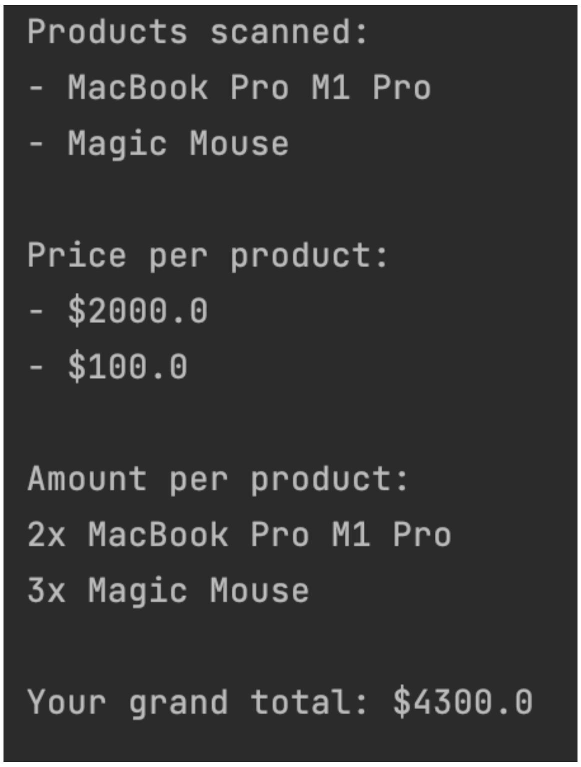
For task 1, the program is a point of sale system that allows a cashier to scan products, calculate prices and discounts, receive payments from customers, and print receipts. At bottom, the key layers that comprise the program are these three layers: controller, model, and integration. The controller layer handles the communication between the view and the model layers. The model layer contains the business logic and data structures of the program. The integration layer contains classes that represent external systems such as inventory system, accounting system, and receipt printer. In addition, there is a view class that simulates a fictional sale process. The view class creates a controller object and calls its methods to start a sale, scan products, pay for the sale, and end the sale. The view class also displays some information about the products and the total amount of the sale. The result shows that the program can perform a basic sale process and display some relevant information to the cashier. The results also show that the program can handle scanning multiple products and calculating the total amount correctly.

Task 2

The unit tests are written using JUnit 5 as the testing framework. The unit tests cover the main functionality and logic of each class and method, and use assertions to verify the expected outcomes. The code provided in the git repository contains several unit test classes for different classes in the model and integration layers. Unit test class has a @BeforeEach method that initializes the objects needed for testing before each test method, and has a @Test annotation to mark it as a test case. The test methods uses methods from the org.junit.jupiter.api.Assertions class to compare the actual and expected values of the methods under test. For example, assertEquals, assertDoesNotThrow, assertNotNull, etc. The code also contains some empty test methods that need to be implemented. For example, in the InventorySystemTest class, there are two empty test methods: testUpdateSaleQuantity and testItemDTORetrievedFromDb. These methods tests the functionality of the updateSaleQuantity and getProductDTOFromDB methods in the InventorySystem class.

Link to the git repository

`https://github.com/danaghafour/iv1350/tree/main/POS-IV1350`



```
Products scanned:
- MacBook Pro M1 Pro
- Magic Mouse

Price per product:
- $2000.0
- $100.0

Amount per product:
2x MacBook Pro M1 Pro
3x Magic Mouse

Your grand total: $4300.0
```

Figure 3.1: A receipt for buying two MacBook Pro M1 Pro and three Magic Mouse from a store.

4 Discussion

I made an effort to use descriptive names for variables, methods, and classes in order to make my code as readable and understandable as possible. Indentation, spacing, naming, and documentation were all done according to Java code conventions. I used comments to describe each class's and method's purpose and functionality, but I refrained from commenting inside methods to avoid cluttering the code.

I prevented the creation of duplicate code by employing helper methods whenever necessary. For instance, in the @BeforeEach method of each test class, I used a helper method to initialize the objects necessary for testing. I put in a concerted effort to write modular code in order to keep my methods and classes concise and to the point. Along with using appropriate levels of abstraction and decomposition, I also divided difficult tasks into manageable chunks. For instance, in the integration layer, I used a different class for each external system, and in the test classes, I used a different method for each test case.

To achieve better encapsulation and reusability, I used objects whenever possible in place of primitive data or static members. For instance, rather than using arrays of primitive data types, I used an ArrayList of Product objects to store the current items in a sale. Additionally, I tried to steer clear of using static members unless they were utility methods or constants that were not dependent on any instance variables.

By using objects as parameters rather than individual values, I was able to avoid having methods with excessively long parameter lists. For instance, rather than passing each attribute of the sale separately to the updateSaleQuantity method in the InventorySystem class, I used a SaleDTO object as a parameter.

With the use of objects as parameters rather than individual values, I was able to avoid having methods with excessively long parameter lists. For instance, rather than passing each attribute of the sale separately to the updateSaleQuantity method in the InventorySystem class, I used a SaleDTO object as a parameter.

The rule of "one comment per public declaration and no comments inside methods" was adhered to by me, and I used Javadoc comments for each public declaration in my code. Additionally, unless it was absolutely necessary to explain some convoluted or obscure logic, I tried to refrain from commenting inside methods.

By dividing the concerns of various program components, I adhered to the MVC and Layer patterns. The View class, which simulates the cashier's interaction with the system, is present in the view layer. The Controller class, which manages communication between the view and model layers, is found in the controller layer. The program's business logic and data structures, such as Sale, Product, Payment, etc., are found in the model layer. Classes representing external systems, such as InventorySystem, AccountingSystem, ReceiptPrinter, etc., are found in the integration layer.

I created my classes and methods with distinct responsibilities and boundaries in a bid to achieve low coupling, high cohesion, and good encapsulation. Where appropriate, I used abstract classes and interfaces to cut down on component dependencies. Additionally, I provided public getters and setters when necessary and used private access modifiers for instance variables. For some methods that were only used within the same package, I additionally used package private access modifiers.

For my unit tests, I used JUnit 5 as the testing framework. I annotated various sections of my test classes with symbols like @Test, @BeforeEach, @AfterEach, etc. To make assertions in my test methods, I also used methods from the org.junit.jupiter.api.Assertions class. I used assertions that contrasted the actual and expected values of the methods under test to ensure that every one of my tests could evaluate themselves. Along with the SUT, I also included my tests in the same package under a different directory called test. I could use package private methods from the SUT in this manner without making them available to other packages.

I created a test class for every class I tested in my program, as instructed. As an example, I created the SaleTest class to test the Sale class.

I tried to cover all branches of if-statements in my tests by using different inputs and scenarios that would trigger each branch. For example, in the SaleTest class, I tested the addProductToSale method with both existing and new products to cover both branches of the if-statement that checks if the product is already scanned. I also used assertions to verify that the expected outcomes were achieved for each branch.