Seminar 4

Object-Oriented Design, IV1350

Dana Ghafour, danagf@kth.se

May 22nd, 2023

Contents

1 Introduction

This report presents the work and results of seminar four in the course Object-Oriented Design (IV1350). The main objectives of this seminar were to learn and apply exception handling and design patterns in a Java program that simulates a point-of-sale (POS) system. 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

Exception handling is a technique that allows a program to deal with errors or unexpected situations without crashing or terminating abruptly. It involves using try-catch-finally blocks, throw and throws statements, and custom exception classes.

Task 2

Design patterns are reusable solutions to common problems in software design. They provide a template or a guideline for how to structure and organize the code.

2 Method

Task 1

To implement exception handling, the following steps were followed:

1. Possible sources of errors or exceptions in the program were identified, such as invalid input, database connection failure, or missing product ID.
2. Which exceptions should be handled by the program and which should be propagated to the caller or the user interface were decided.
3. Try-catch-finally blocks were used to enclose the code that might throw an exception and handle it appropriately in the catch clause. For example, I used a try-catch block to get a product from the inventory system and catch any ProductIDMissingException or InventoryDBException that might occur.
4. Throw statements were used to create and throw new exceptions when needed. For example, I used a throw statement to create and throw a new OperationFailedException when both the inventory system and the developer log were unavailable.
5. Throws statements were used to declare which exceptions a method might throw to its caller. For example, I used a throws statement to declare that the scanProduct method might throw an OperationFailedException.
6. Custom exception classes were created to represent specific errors or situations relevant to the program domain. For example, I created a ProductIDMissingException class to represent the situation when a product ID was not found in the inventory system, and an InventoryDBException class to represent the situation when the inventory system server was down.

Task 2

To implement the chosen patterns, the steps were as follows:

1. We identified the scenarios that could benefit from using a design pattern. For example, the total revenue view and file output needed to be updated when a payment was made, and the accounting system and inventory system needed to have only one instance each.
2. We chose a suitable design pattern that could solve the problem or scenario. For example, we chose the Observer pattern to decouple the payment from its observers and notify them of any changes, and we chose the Singleton pattern to ensure that there was only one instance of the accounting system and inventory system classes.

3. The design pattern was implemented according to its structure and participants. For example, the Observer pattern was implemented by creating a PaymentObserver interface and making the TotalRevenueView and TotalRevenueFileOutput classes implement it. We also made the Payment class maintain a list of payment observers and notify them when a payment was made. The Singleton pattern was implemented by creating a private static field to store the instance of the AccountingSystem and InventorySystem classes and a public static method to access it. Their constructors were also made private to prevent other classes from creating new instances.

3 Result

As shown in figure 3.1, the program simulates two sales with different products and payments. The program also handles exceptions by printing informative messages to the console or to a log file. For example, when a product ID is not found in the inventory system, the program prints ”Product ID was not found in the inventory system.” and continues the sale. When the inventory system server is down, the program prints ”Inventory system server is down.” and terminates the program. The program also prints the total revenue to the console and to a file after each payment.

Task 1

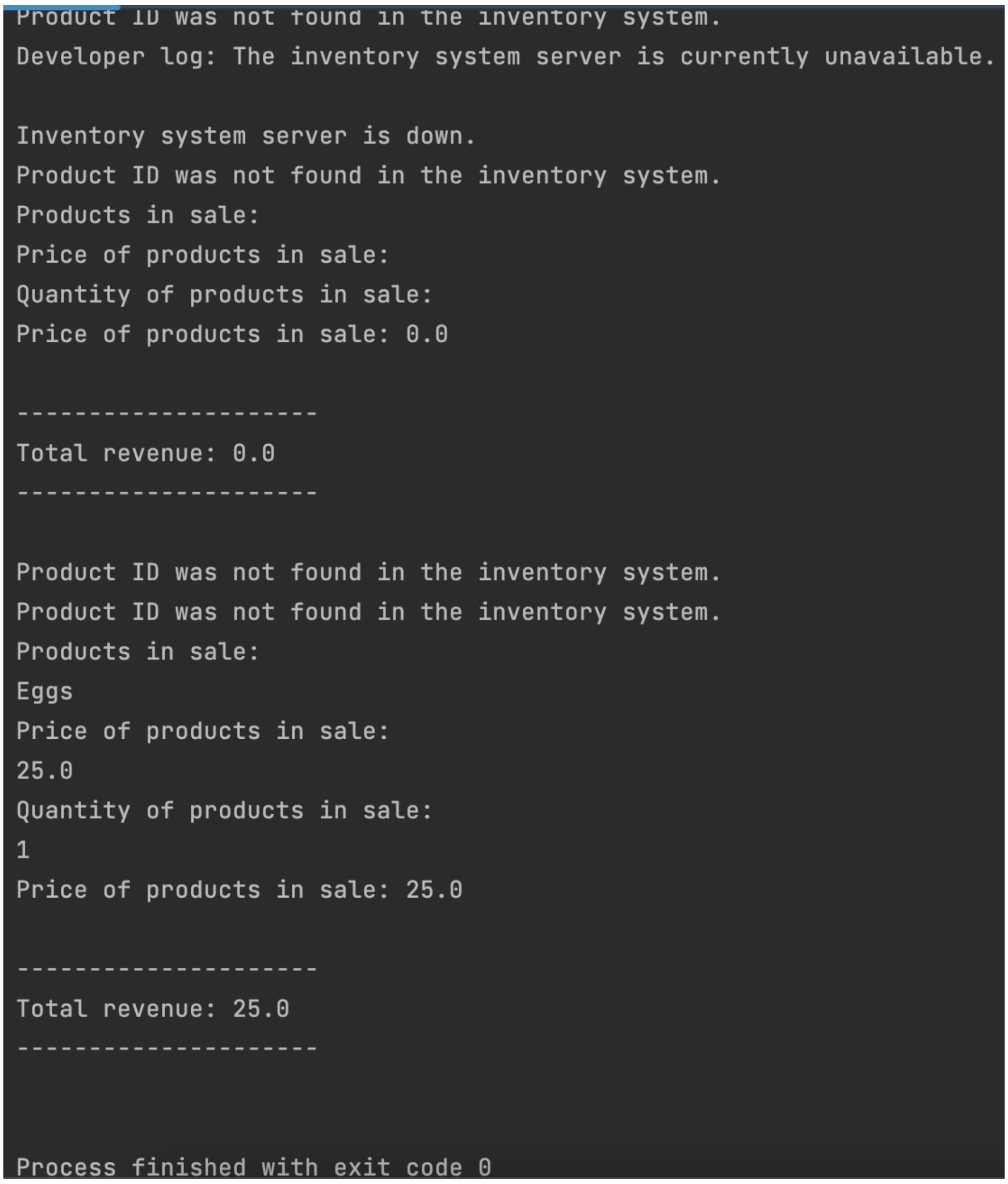
The program uses exception handling to deal with errors or unexpected situations, such as missing product ID or inventory system server down. A sample run of the program is shown in 3.1.

Task 2

The program is a point of sale system that simulates a sale process with products, payment, and receipt. The program uses the Observer pattern to update the total revenue view and file output when a payment is made. A sample run of the program is shown in 3.1. A non-exhaustive list of the most central classes for this task:

- Payment represents a payment made by a customer. It maintains a list of payment observers and notifies them when a payment is made. It also implements a method to add new observers to the list.
- PaymentObserver, a interface that represents an observer that observes customer payments. It defines a method to update the observer when a new payment is made.
- TotalRevenueView, a view that displays the total revenue. It implements the PaymentObserver interface and updates its output when a new payment is made.
- TotalRevenueFileOutput represents a file output for the total revenue. It implements the PaymentObserver interface and updates its output when a new payment is made.
- Controller is responsible for controlling the POS system flow and interacting with external systems. It adds a new observer for the CustomerPayment class to the controller.

- View represents the cashier's view. It creates an instance of TotalRevenueView and adds it as an observer to the controller.



```
Product ID was not found in the inventory system.
Developer log: The inventory system server is currently unavailable.

Inventory system server is down.
Product ID was not found in the inventory system.
Products in sale:
Price of products in sale:
Quantity of products in sale:
Price of products in sale: 0.0

---------------------
Total revenue: 0.0
---------------------

Product ID was not found in the inventory system.
Product ID was not found in the inventory system.
Products in sale:
Eggs
Price of products in sale:
25.0
Quantity of products in sale:
1
Price of products in sale: 25.0

---------------------
Total revenue: 25.0
---------------------



Process finished with exit code 0
```

Figure 3.1: A sample run of the program.

Link to the git repository

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

4 Discussion

Exception handling

The code meets some of the best practices for exception handling from chapter eight. It uses checked and unchecked exceptions appropriately, depending on whether the caller can recover from the error or not. For example, it uses a checked exception (OperationFailedException) for the scanProduct method in the Controller class, because the caller (View class) can handle the exception and continue the program. It also uses an unchecked exception (ProductIDMissingException) for the getProductDTOFromDB method in the InventorySystem class, because the caller (Controller class) cannot recover from the error and should terminate the program. The code also names the exceptions after the error condition, such as ProductIDMissingException and InventoryDBException. This makes the exceptions more descriptive and meaningful. The code also includes information about the error condition in the exception message, such as the product ID that is missing or the inventory system server that is unavailable. This helps to identify and debug the error. The code also uses functionality provided in java.lang.Exception, such as super(message) and super(message, cause), to create custom exceptions that inherit from RuntimeException or Exception. The code also writes javadoc comments for all exceptions, explaining what they represent and when they are thrown. This helps to document and communicate the exceptions to other developers and users. The code also notifies users and developers when an exception is caught, by printing an informative message to the console or writing an error report to a log file. This helps to inform and alert the users and developers about the error and its consequences.

Design patterns

The code implements two design patterns: Observer and Singleton. The Observer pattern is used to update the total revenue view and file output when a payment is made. This pattern decouples the subject (Payment class) from its observers (TotalRevenueView and TotalRevenueFileOutput classes), allowing them to vary independently. This pattern also allows adding new observers without modifying existing code, following the open-closed principle. The Singleton pattern is used to ensure that there is only one instance of the AccountingSystem and InventorySystem classes. This pattern ensures that these classes have a single point of access and control, preventing inconsistent states or conflicts. This pattern also reduces memory usage and improves performance by avoiding creating multiple instances of these classes.

Observer and observed object

In my opinion, the choice of the observer and observed object for the Observer pattern is reasonable. The Payment class is a suitable observed object because it changes its state when a payment is made. The TotalRevenueView and TotalRevenueFileOutput classes are suitable observers because they need to update their output when a payment is made.

Reference passing

The reference to observer is passed to the observed object by calling the addPaymentObserver method on the Payment class. This method takes a PaymentObserver object as a parameter and adds it to a list of payment observers. This does not have bad effects on coupling or cohesion because it follows the dependency inversion principle. The Payment class depends on an abstraction (PaymentObserver interface) rather than a concrete implementation (TotalRevenueView or TotalRevenueFileOutput classes). This increases cohesion within classes and reduces coupling between them.

Data passing

The data passed from observed object to observer is the amount for sale as a double value. This does not break encapsulation of the observed object because it does not expose any private fields or methods of the Payment class. It only passes a public value that represents its state.