Seminar 4

Object-Oriented Design, IV1350

Ayda Ghalkhanbaz
aydag@kth.se

May 19, 2024

Contents

1 Introduction

The fourth task of the project includes handling exceptions, and adding design patterns such as factory, singleton and strategy into the existing Java code from the previous task.

This report covers the implementation details of exceptions and chosen pattern designs for this seminar, the results then will be shown and discussed in the later sections. The code is written in Java using Visual Studio Code. I cooperated with Naveed Rahman and Nora Say to accomplish this task.

2 Method

As a preparation to complete this seminar, chapter 8 and 9 of the book “A First Course in Object Oriented Development” were thoroughly examined as well as the video lectures.

2.1 Task 1

Handling the desired exceptions in the task starts with determining the class and further the method that can possibly throw an exception (such as Item Not Found or Database Failure). Since the used exceptions are not in-built in Java, two child classes of Exception are implemented that will represent the new desired exception types .

After implementing the actual exceptions and modifying the source of the exceptions, all other methods that call the source method are modified so that they also handle the exceptions. Finally, try-catch blocks were added to the View class to handle the possibly thrown exceptions.

The exception error messages are then printed both on screen (terminal) and to a file log using the FileLogger class.

As the second step, some unit tests were written to approve the functionality of the exceptions.

2.2 Task 2

Since the second task included implementing several design patterns, the use of each pattern and where to implement them were determined first.

The first design pattern, namely Observer, is implemented to notify any new changes to the total income of the store.

The second design pattern, Strategy, is implemented for the discount calculations since there are two different algorithms to calculate the discount. For simplicity, another design pattern called Factory is implemented to simplify the decision of choosing an appropriate algorithm to calculate the discount.

Finally, the singleton pattern was implemented to avoid unnecessary initialization of factory instances. In this way, the entire program can access one and only one factory instance.

3 Result

The code can be found on: GitHub.

Navigate through: `pos-src-main-java-se-kth-iv1350` to find the program. To find tests, navigate through: `pos-src-test-java-se-kth-iv1350`. I know it is a very long path:)

3.1 Task 1

The result of running a fake simulation is shown in figure 3.1. The exception messages are also written to a file which can be seen below.

.... The exception log using FileLogger ....

```
An exception was thrown at 2024-05-19 11:18:29 :
< Something went wrong with server connection. Please try again later! >

An exception was thrown at 2024-05-19 11:18:29 :
< Item with item ID 6 not found! >
```

The program throws an ItemNotFound exception whenever the cashier scans an item with an invalid item ID (an ID that is not in the inventory system) or scans an item with item ID 4 (which is hardcoded), in this case an exception of DatabaseFailure will be thrown. As mentioned earlier, in the catch block the error messages are both shown to the user and the developer by printing them out on the screen and a log file.

The source method that throws these exceptions is the getItemInfo() method in InventorySystem. These exceptions are handled in the View where the cashier actually scans items and the controller fetches the item information from the inventory system.

Some tests are also added into the InventorySystemTest and ControllerTest classes. Methods in the InventorySystem and Controller are tested using assertThrows to check whether a correct type of exception has been thrown.

3.2 Task 2

As it is shown in figure 3.1, a notification was sent to all observers saying that the income of the store has changed. The main changes for this part of the task includes creating a new interface called SaleObserver which classes TotalRevenueView and TotalRevenueFileOutput implement. These classes have one main job which is updating the total revenue based on the total price of a sale. These changes are then notified to the observers (which in this case two observers, one in view/screen and one in form of file output).

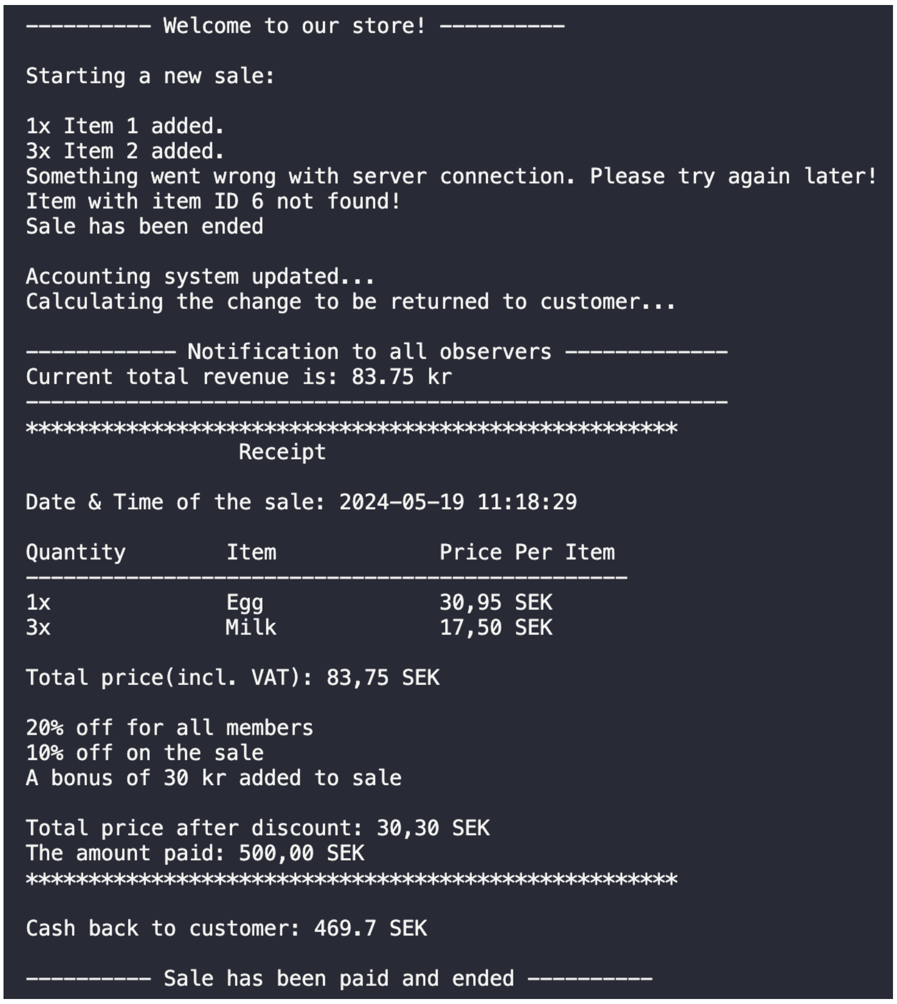
Other changes corresponding to the Observer can be found in the Sale class, where the actual notifications are sent to the observers in conjunction with a change in total price. Also, new observers can be added to the observer list, which is done by the controller.

Additionally, other design patterns such as Strategy, factory and singleton have been implemented. Main changes in connection to these patterns are as follows:

- For the strategy pattern, a new interface called DiscountCalculator is created, which later the classes DiscountAmount and DiscountPercent implement as two different algorithms for calculating the discounts. The responsible methods in Sale and Controller classes for adding discounts to the sale are modified after the changes.

- The Factory design pattern is implemented within its own class called DiscountFactory and is responsible for choosing an appropriate algorithm for the client. The instance of the Factory is used in the Sale class to get a discount calculator algorithm to modify the amount of sale for the customer.

- The singleton pattern is added to the DiscountFactory so that only one instance of Factory is used during the entire program. This is done by creating a private static DiscountFactory instance in the class as well as a public static method that returns the same instance whenever it is called (for example in the Sale class within the reduceSale method).



```
---------- Welcome to our store! ----------

Starting a new sale:

1x Item 1 added.
3x Item 2 added.
Something went wrong with server connection. Please try again later!
Item with item ID 6 not found!
Sale has been ended

Accounting system updated...
Calculating the change to be returned to customer...

------------ Notification to all observers -------------
Current total revenue is: 83.75 kr
--------------------------------------------------------
****************************************************
                 Receipt

Date & Time of the sale: 2024-05-19 11:18:29

Quantity        Item             Price Per Item
------------------------------------------------
1x              Egg              30,95 SEK
3x              Milk             17,50 SEK

Total price(incl. VAT): 83,75 SEK

20% off for all members
10% off on the sale
A bonus of 30 kr added to sale

Total price after discount: 30,30 SEK
The amount paid: 500,00 SEK
****************************************************

Cash back to customer: 469.7 SEK

---------- Sale has been paid and ended ----------
```

Figure 3.1: The output of the fake system simulation

4 Discussion

The Exception handling meets the best practices mentioned in the task. The checked exception is used for the case when an item ID doesn't exist in the database since this exception is due to the business logic and not a programmatic error. On the other hand, the DatabaseFailure is considered as an unchecked exception and is thrown whenever the cashier scans an item with item ID 4.

These exceptions are thrown within the integration layer (InventorySystem) and are handled by the view later using try-catch blocks. Since the Controller fetches item information from the inventory system, it also may throw an exception of types ItemNotFound or DatabaseFailure.

Additionally, the name of these exceptions clearly specifies their purpose and they also hold a descriptive message in case of exception which explains the error for both users and developers.

From the provided code we can see that the exception classes implement the `java.lang.Exception` and use the provided functionality such as super(message). Moreover, the exception classes are documented with descriptive comments that clarifies the functionality and usage of each public method.

As mentioned in earlier sections, there are unit tests written to approve the functionality of the exception handlers using assertThrows. This assertion checks whether the expected exception type is the same as the thrown exception, and in case of failure it prints out a test failure message. Since all the written tests pass, we can conclude that the exception handling works as intended. Thus, we can approve that the exception handling follows all mentioned best practices bullets in the task.

The program includes four different design patterns: Observer, Strategy, Factory and Singleton.

The Observer pattern is used to notify all observers that have subscribed for any changes to the revenue of the store. In other words, whenever the total revenue is changed by the subject (Sale class), all subscribed observers (TotalRevenueView and TotalRevenueFileOutput) will be notified about it. The pattern is independent of the number of observers. As long as the observers implement the SaleObserver interface they will get a notification about the changes and this doesn't violate any rules to make the program high coupled, low cohered or not encapsulated.

The passed data from the subject to the observers is the total price of sale which is of type double and doesn't violate any encapsulation rules.

The Strategy pattern is used to create various algorithms for the same job, which allows the client to choose an appropriate one among all. The discount calculation was chosen to be designed using this pattern since there are two different ways of discount calculations according to the business logic of the store. The first algorithm calculates the discount with the consideration that the discount is an amount to be deducted from the total price. While the other algorithm is used whenever the discount is a percentage that should be deducted.

The Factory pattern simplifies the choice of the appropriate algorithm for the client by examining whether the type of discount is Amount or Percent. It will then pick the right algorithm and send it back to the client.

The Singleton pattern is used to ensure that there is only one instance of the DiscountFactory class. This pattern makes sure that the DiscountFactory class has only one point of access and control which leads to a consistent state without any conflicts.