Seminar 4

Object-Oriented Design, IV1350

Viktor Jäger, vjager@kth.se

May 13, 2020

Contents

1 Introduction

The fourth seminar in the course Object-Oriented Design aims to focus on handling Exceptions and Design Patterns. The mandatory tasks for this seminar was to implement exception handling in the program as well as an implementation of the Observer pattern to provide a display that shows the total amount paid for purchases since the program started. On top of this, unit tests had to be implemented for the exception handling. This report will provide a description of the method used to develop the exception handling and implementation of the design pattern, as well as the tests written to verify the functionality. The discussion will evaluate the result and discuss the most interesting decision and insights made during the implementation. To gain further understanding and knowledge when designing the program I discussed my solutions with my classmate Vera Lindström. Having the chance to discuss my ideas and solutions with another person have resulted in improvements in the final design and made it possible to consider different approaches to certain problems.

2 Method

2.1 Exception handling

The methodology for implementing the exception handling in the program are based on the theory found in chapter 8.2 in the book *A First Course in Object Oriented Development: A Hands-On Approach* by Lars Lindbäck. The most important things to consider while implementing exceptions can be summarized in the following bullet points:

- Only use exceptions for error handling. Using other methods for error handling leads to low cohesion since the return value of a method will be used for two different purposes (one for the expected outcome and one for the error handling).
- Decide between checked and unchecked exceptions, where checked exceptions should be used for business logic errors and unchecked exceptions should be used for programming errors.
- Using the correct abstraction level for exceptions, where exceptions in lower layers may have to be wrapped into more generalised exceptions if they make little to no sense to higher layers in the program.
- Naming the exceptions after the error conditions and also include information about error condition that caused the exception. The name should clarify why the exception was thrown and preferably end with 'Exception' in order to make it easy to recognise. The information included in the exception should consist of all the data that together make up the exceptional condition as well as a string containing an explanatory message of the exception that is intended for developers or administrators to trace the problem.
- Do not change the state of an object if an exception is thrown. This can be solved by placing operations that could fail before the operations that alter the state of an object, checking parameter values before changing any states. It is good practice to make objects immutable in order to prevent unexpected behaviour.
- Notify users and developers in appropriate ways. The messages to the user should be carefully considered and consistent in order to not cause any confusion. The messages to the developers and administrators should be detailed so that the exception can be traced and dealt with, for example with an exception stack trace. Those messages should be printed to a log from a dedicated log handling component, in order to keep the error logging consistent.

- Write javadoc comments for all exceptions and write unit tests for all classes that uses exceptions.

All points in the above list have been taken into consideration while implementing the exception handling in the program.

2.2 Design Patterns

The implemented pattern in this program are the Observer pattern, its purpose is to notify observers when an observed class changes state. The observer pattern allows the lower layers to call upper layers without any dependency from lower to upper layers, which would otherwise not be possible without ruining the MVC if the pattern were to be omitted. The observer pattern consists of an interface which is placed in the layer of the class that is to be observed. Classes that are interested in observing the class implements the observer interface and their references are passed down to the class that are to be observed. The observed class adds all the observers to a list of observers and then notifies them when the observed class changes the state of interest. By using an interface this is as previously mentioned done without any dependencies from lower layers to higher layers. The specific solution to the implementation of the observer pattern is presented in the result.

3 Result

The whole project, including exception handling and design patterns, can be found at `https://github.com/ViktorJager/IV1350-Exceptions-and-Design-Patterns`

3.1 Exception handling

The final program is a refined version of the previous implementation created during seminar three. While a few alterations and additions have been made, it mostly remains the same with the addition of exception handling for cases where a specified item identifier is not found and exception handling for database failure. The exception handling for alternative flow 3-4a in Process sale requirements specification is found in the integration layer and thrown when the controller requests an invalid item identifier from the simulated database. The exception is considered checked and the same exception passed up to and caught in the view and an explanatory message is presented to the ui. The program prints the error message with the faulty identifier and asks the user to enter a new item identifier. Since this exception is the expected outcome, it is not logged to the log. The database connection failure is simulated so that when the hard-coded item identifier '999999' is searched for, an exception 'DatabaseFailureException' is thrown. This is considered an unchecked exception and is caught in the controller, since it is not suitable for the view. The exception is wrapped to the more general 'OperationFailedException' and thrown to the view, where it is caught, notifies the user of a failed operation. The exception is also logged to a log with a timestamp, message and an exception stack trace. Unit tests for the classes that handles exceptions can be found in the git repository.

3.2 Design Patterns

An observer interface called 'SaleObserver' was added to the model which is implemented by the class 'TotalRevenueView' in the view layer. The view creates and passes the reference of the 'TotalRevenueView' when the view is instantiated to the controller which passes the reference further to the 'Sale' instance. 'Sale' is the class that is to be observed so it stores all the observers in a list and when a sale is paid for, it calls the method 'notifyObservers'. This method takes an amount as a parameter that represents the amount paid for the completed sale and notifies all the observers. When 'TotalRevenueView' is notified, its override method 'newPayment' adds the amount for the completed sale to the total revenue and prints the total amount paid for purchases since the program started to the console.

```
------------------------------------------------------------------------
Building RetailPOS 1.0
------------------------------------------------------------------------

--- exec-maven-plugin:1.5.0:exec (default-cli) @ RetailPOS ---
A new sale has been started.
Price: 250, Name: Firebird, Quantity: 1, Total Price: 250
ERROR | 13 May 2020, 02:47:31 | Could not register item
Price: 250, Name: TeeShirt, Quantity: 1, Total Price: 500
Invalid item identifier: '999'
Price: 250, Name: Firebird, Quantity: 2, Total Price: 750
ERROR | 13 May 2020, 02:47:31 | Could not register item
Price: 238,5, Name: Destroyer, Quantity: 1, Total Price: 988,5
Invalid item identifier: '123'
Price: 238,5, Name: Destroyer, Quantity: 2, Total Price: 1227
Price: 125, Name: AirMax2020T, Quantity: 1, Total Price: 1352
Price: 125, Name: AirMax2020T, Quantity: 2, Total Price: 1477
Invalid item identifier: '4'
The sale has ended.
Total price: 1477.0
***** Total amount for all completed sales: 1477.0 *****
RECEIPT
Retail Store X
UnknownStreet 13, 75441, Uppsala

2020-05-13 02:47:31

Name:           Quantity:       Price:
Firebird                2       500
TeeShirt                1       250
Destroyer               2       477
AirMax2020T             2       250

Total price incl. VAT: 1477
Total VAT: 227

Amount paid: 1750
Change: 273

------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.138 s
Finished at: 2020-05-13T02:47:31+02:00
Final Memory: 7M/34M
------------------------------------------------------------------------
```

Figure 3.1: Printout of a sample run with hard-coded input, exception handling and an observer

4 Discussion

My solutions to the tasks in seminar 4 are heavily based on the code provided in the git repository from the course material. With that in mind, I have still tried to carefully consider my solutions and made sure that they are correctly implemented. This was mainly done by studying the bullets points in chapter 8.2 and implementing them after those criterias. I argue that my implementation of the exception handling follows the points provided in the task specification for the seminar 4. Checked exceptions extends Exceptions while unchecked exceptions extends RuntimeExceptions. The database failure exception is wrapped to a more general exception before its thrown to the view in order to achieve right abstraction levels for the exceptions. Naming and information about exceptions are included and they notifies the users and developers with error messages and logging when necessary. No objects changes states if an exception is thrown, it could be argued that parameter checking could be added for more rigidity in the system. An argument against that is the only passed parameter that causes exceptions in the current implementation are a single string in all methods. If the string is null or any non valid item identifier, an ItemNotFoundException will be thrown which already implies that something is wrong with the input. If the program were to be developed further, this would probably be an added feature though. In addition to this, the exceptions use functionality provided by java.lang.Exceptions and unit tests are implemented. The tests may seem a bit lackluster but they are specifically designed for the seminar 4, and does not include testing of public and package private methods that does not handle exceptions. The unit testing handles the most common cases but there may still be some edge case that are not fully covered.

I tried to follow the provided examples and keep the observer pattern simple and straightforward in order to reduce confusion and make it a correct implementation of the pattern. As explained in the result I create the observer 'TotalRevenueView' in the view when it is instantiated, it is passed through the controller to the sale instance whenever a new sale is started. This is done in order to keep low coupling, by not introducing any new dependencies between layers. The data sent from the observed object are the total amount paid for each sale which is encapsulated in an 'Amount' class. The implemented interface is called 'SaleObserver' in order to be more general and allow more added features in the future that includes other kinds of observers to the class. Since the sale class is quite central in my implementation I thought it made sense to not make the observer interface too specific.