# Seminar 4 - Exceptions and Design Patterns

## Object-Oriented Design, IV1350

Alexander Lundqvist

Alexlu@kth.se

2021-05-24

# Contents

# 1 Introduction

This seminar is split into two areas. The first one is to implement exception handling for the program that was constructed in the previous seminar. The second area involves implementing additional functionality to the program by implementing the observer pattern.

## 2 Method

For task 1 I implemented two main exceptions for dealing with both an unchecked and checked exception. The checked exception involved in handling errors regarding wrong item identifiers and it was implemented in the InvalidItemIDException class. As for the unchecked exception I added the InventoryFailureException class that handles database related errors. The main reason for choosing these types of exceptions is that they are the most relevant regarding the alternative flow 3-4a in the Requirements Specification for Process Sale. They are also one of the few where the user/cashier actually performs “manual input” and thus it is subject to high risk of error.

To actually implement the classes I had to continuously consult the course literature and several online sources, as I didn’t find the video lectures on this subject to be enough. The most difficult part was to figure out the abstraction chain of the exceptions.

Some additional files were created for the program to work. One example is the OperationFailureException which I didn’t mention earlier. This class acts as a wrapper in the controller for the deeper exceptions. The LogHandler was implemented in the util package as its primary function is to write the exceptions to a file which I feel is a function suited for that package.

In the second task I only implemented the mandatory part, which was to implement the Observer pattern. To do this I implemented the SaleObserver interface which laid the foundation for the TotalRevenueView and TotalRevenueFileOutput classes. The classes were specified in the seminar task so I didn’t have much say in the reason for them.

Lastly I implemented unit test for the exception classes except the OperationFailureException class. The observer classes did not get any tests.

## 3 Result

For the first task the following files were constructed.

https://github.com/KrimReaper/IV1350-POS/blob/main/posSem4/src/main/java/se/kth/iv1350/pos/integration/InvalidItemIDException.java

https://github.com/KrimReaper/IV1350-POS/blob/main/posSem4/src/main/java/se/kth/iv1350/pos/integration/InventoryFailureException.java

https://github.com/KrimReaper/IV1350-POS/blob/main/posSem4/src/main/java/se/kth/iv1350/pos/util/LogHandler.java

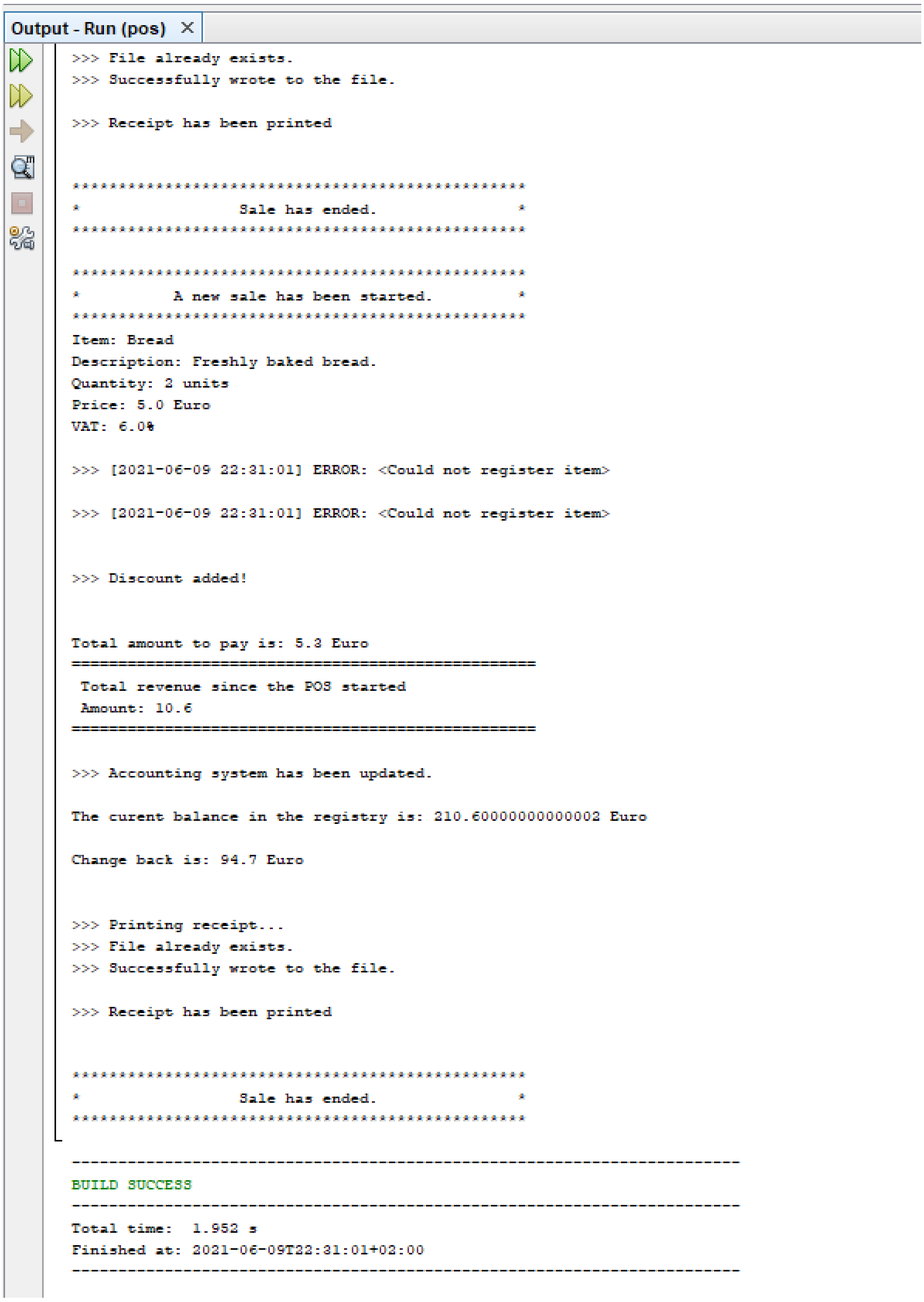
For the second task the following main classes were added

https://github.com/KrimReaper/IV1350-POS/blob/main/posSem4/src/main/java/se/kth/iv1350/pos/model/SaleObserver.java https://github.com/KrimReaper/IV1350-POS/blob/main/posSem4/src/main/java/se/kth/iv1350/pos/view/TotalRevenueView.java

https://github.com/KrimReaper/IV1350-POS/blob/main/posSem4/src/main/java/se/kth/iv1350/pos/util/TotalRevenueFileOutput.java

I made a lot of other changes to the whole program to accommodate for these new classes. As such the whole program is too extensive to add here, please check the github repository instead: https://github.com/KrimReaper/IV1350-POS/tree/main/posSem4

Lastly is an image describing the console output of the program. To clarify, the image depicts the last sale made as I could not get a detailed image if I tried to include everything. But the program performed two identical sales, so you can see that the TotalRevenueView is displaying the correct amount.


Output - Run (pos)

```
>>> File already exists.
>>> Successfully wrote to the file.

>>> Receipt has been printed


**************************************************
*                 Sale has ended.                *
**************************************************

**************************************************
*          A new sale has been started.          *
**************************************************
Item: Bread
Description: Freshly baked bread.
Quantity: 2 units
Price: 5.0 Euro
VAT: 6.0%

>>> [2021-06-09 22:31:01] ERROR: <Could not register item>

>>> [2021-06-09 22:31:01] ERROR: <Could not register item>


>>> Discount added!


Total amount to pay is: 5.3 Euro
===================================================
 Total revenue since the POS started
 Amount: 10.6
===================================================

>>> Accounting system has been updated.

The curent balance in the registry is: 210.60000000000002 Euro

Change back is: 94.7 Euro


>>> Printing receipt...
>>> File already exists.
>>> Successfully wrote to the file.

>>> Receipt has been printed


**************************************************
*                 Sale has ended.                *
**************************************************

------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.952 s
Finished at: 2021-06-09T22:31:01+02:00
------------------------------------------------------------------------
```

# 4 Discussion

The creation of the exceptions had to follow nine points regarding *exception handling best practice*s and to my understanding I have followed them all in my work. Most of these point I won't mention as they are trivial such as naming convention and information included in the errors, as they are already illustrated in the result chapter. However I will mention a few of them.

When implementing the exceptions I had to consider the abstraction level. When I first started I had no understanding at all where to put the exceptions for the abstraction level to be correct. As mentioned in the literature and the lectures, best practice is to implement the primary exceptions in the package where they are intended to be thrown. As such I implemented them in the integration layer and added the throw clause to appropriate classes where the errors are most likely to originate from. I also included a wrapper class in the controller layer to abstract the errors originating from the integration layer.

The exceptions are also logged in a separate text file. As I understand we are not supposed to log checked errors, but for this task I decided to do it anyway to help with my debugging.

I realize that the InventoryFailureException could have been named better, but the implementation of the class could be expanded to handle all exceptions regarding the inventory "database".

The only design pattern that was implemented was the Observer pattern, as it was the minimum requirement for task 2, therefore I do not have any personal reason as why it was implemented. I understand the underlying reasoning why you would use the Observer pattern and so, I fully agree with the choice of observer/observables in the seminar task.

As the observers only get passed to the intended observable object to monitor change and the observable object doesn't really have any connection with the class that the observer represents the coupling will be on an abstract level and can be considered minimal.

The only data that is passed into the observers is the running total of the sale. I wouldn't really consider that the Sale class "hands out" its data in this regard. The only thing that Sale actually does is that it tells the observers (Which are also stored as a reference in Sale) that "Hey guys, please update yourselves with X amount". So no, I don't think it actually breaks the encapsulation of Sale.

Furthermore it is almost self explanatory that when implementing observers, and thus separating the layers/classes by abstraction, we can also achieve higher cohesion.