Seminar 4

Object-Oriented Design, IV1350

Ossian Dillner – ossiand@kth.se

2017-05-15

Contents

1 Introduction

In preparation for this seminar the student was given two tasks to perform pertaining to the usage of exceptions and design patterns in object oriented development. The student was also given a choice between performing these tasks on the program they developed in the previous seminar or on a given program independent of previous work, I made the choice of using the later of these options.

The first task consisted of implementing correct exception usage and handling into the program,

The second task consisted of two subtasks, the first one of these subtasks instructed the student to implement the observer design pattern, allowing the student to create a display in the view package that observed the operations of the program without affecting its state in any way.

The second subtask required the student to implement two GoF design patterns of their choosing into the given program.

I choose to perform all of these tasks on my own.

2 Method

- **Task 1**

Before I started the task of implementing proper exception usage and handling into the given program I made sure to first get a good understanding of the programs flow and purpose by carefully reading the source code and Javadoc comments, as well as compiling and running the program with a few different test scenarios.

This careful examination allowed me to deduce how to best implement exception handling following guidelines from the course literature as well as online sources. My deduction led me to the insight that I needed to create two new types of exceptions to achieve the proper level of abstraction.

The first more descriptive with the purpose to be thrown from the model layer to the controller and another that more general purpose that is then thrown from the controller layer to the view, this so that the inner workings of the program are not exposed to a user, preventing the inherit security flaws doing so would bring about as well as reducing user confusion. Lastly, I decided that both of these exceptions should be checked as they are business logic errors and are not caused by bugs in the code.

- **Task 2a**

Following the standard guidelines for implementing the observer pattern I was able to apply said pattern to the program by simply creating an observer interface with the required definitions. This allowed me to create a class in the view package that implemented the observer interface allowing it to silently observe and display the inner operations of the program to the user without interfering or modifying any of the programs inner data or flow.

- **Task 2b**

Unfortunately, because of the redundant nature of trying to implement design patterns into already (almost) functioning classes I made the choice to go with a very simple direction when accomplishing this task. I choose to use the suggested Singleton pattern in the ProductCatalog as well as using the Iterator pattern in the Receipt class.

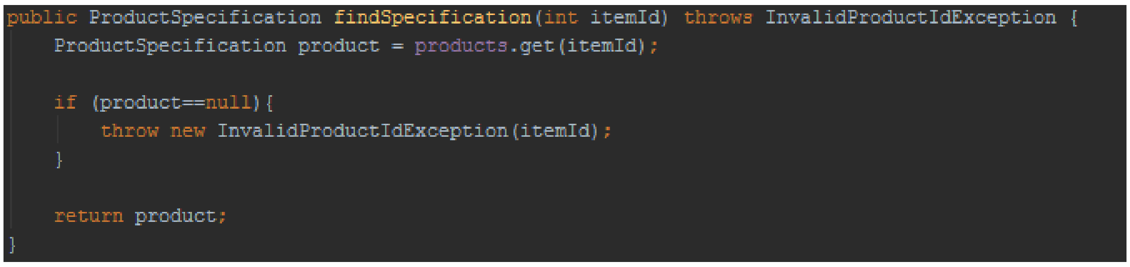
The combination of the simplicity of these chosen patterns in combination with the fact that most of the work had already been done made for a very straightforward and easy implementation of the aforementioned design patterns, barely requiring any changes at all.

3 Result

Github repo: https://github.com/gravestoned/lab4

- **Task 1**

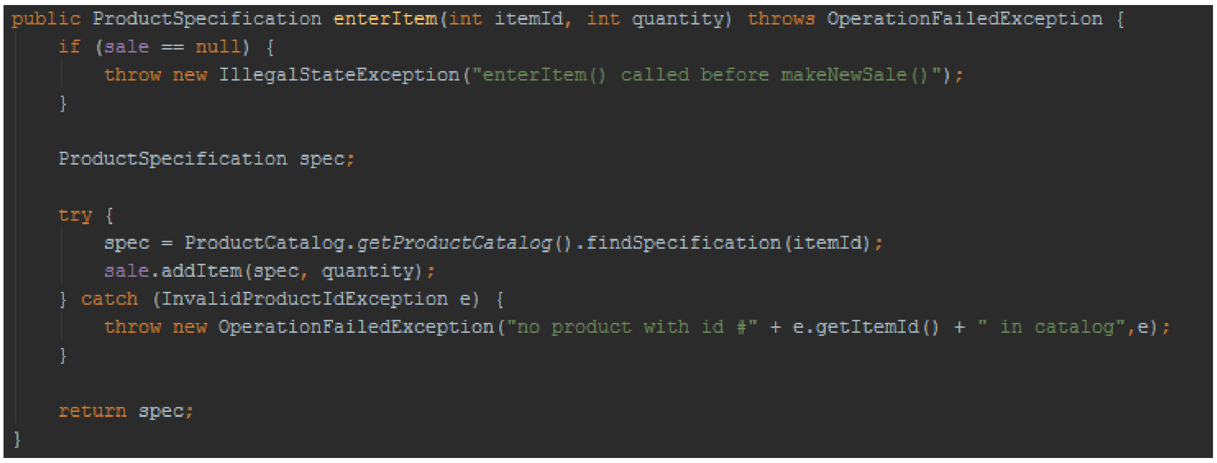
The program is fairly straightforward, it represents a purchase of one or several items. A new sale is initiated with the makeNewSale method in the controller, items are then added to the sale using the enterItem method, finally the sale is finished by calling the makePayment method. The important information pertaining to this task is how the exception handling has been implemented. I decided to add two types of exceptions, InvalidProductIdException and OperationFailedException, the first one is thrown by the ProductCatalog as illustrated in the code excerpt in fig 3.1.



```
public ProductSpecification findSpecification(int itemId) throws InvalidProductIdException {
    ProductSpecification product = products.get(itemId);

    if (product==null){
        throw new InvalidProductIdException(itemId);
    }

    return product;
}
```

Fig 3.1

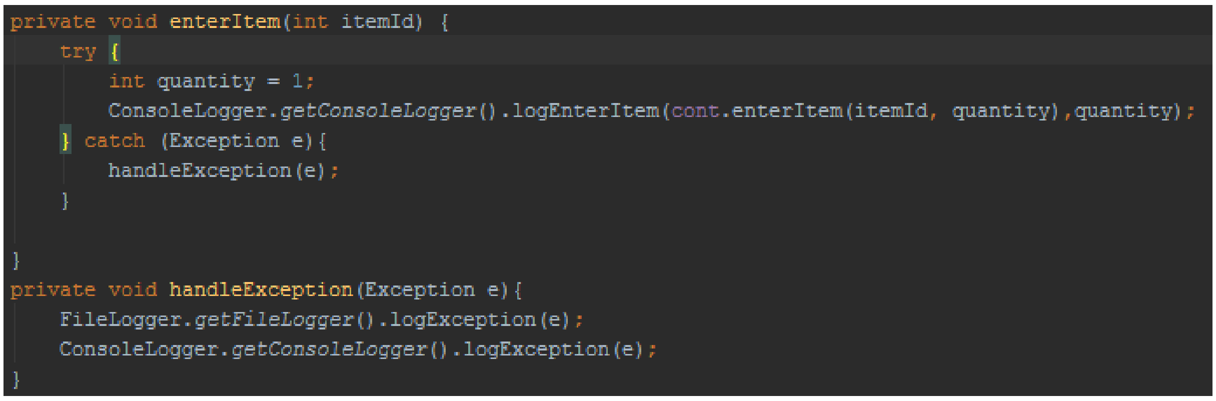
As you clearly see, this exception is thrown when attempting to retrieve a ProductSpecification that does not exist, it is then caught in the controller as illustrated in the code excerpt in fig 3.2



```
public ProductSpecification enterItem(int itemId, int quantity) throws OperationFailedException {
    if (sale == null) {
        throw new IllegalStateException("enterItem() called before makeNewSale()");
    }

    ProductSpecification spec;

    try {
        spec = ProductCatalog.getProductCatalog().findSpecification(itemId);
        sale.addItem(spec, quantity);
    } catch (InvalidProductIdException e) {
        throw new OperationFailedException("no product with id #" + e.getItemId() + " in catalog",e);
    }

    return spec;
}
```

Fig 3.2

The controller then throws an OperationFailedException to the view, containing a message for the user as well as the exception that caused the operation to fail. The view then catches the latter exception and passes it on to the file and console logging classes as illustrated in the code excerpt in fig 3.3.



```
private void enterItem(int itemId) {
    try {
        int quantity = 1;
        ConsoleLogger.getConsoleLogger().logEnterItem(cont.enterItem(itemId, quantity),quantity);
    } catch (Exception e){
        handleException(e);
    }

}
private void handleException(Exception e){
    FileLogger.getFileLogger().logException(e);
    ConsoleLogger.getConsoleLogger().logException(e);
}
```

Fig 3.3

- **Task 2a**

The implementation of the Observer design pattern was fairly simple and didn't require much modification of the given program, the implemented changes are illustrated in the diagram in fig 3.4.

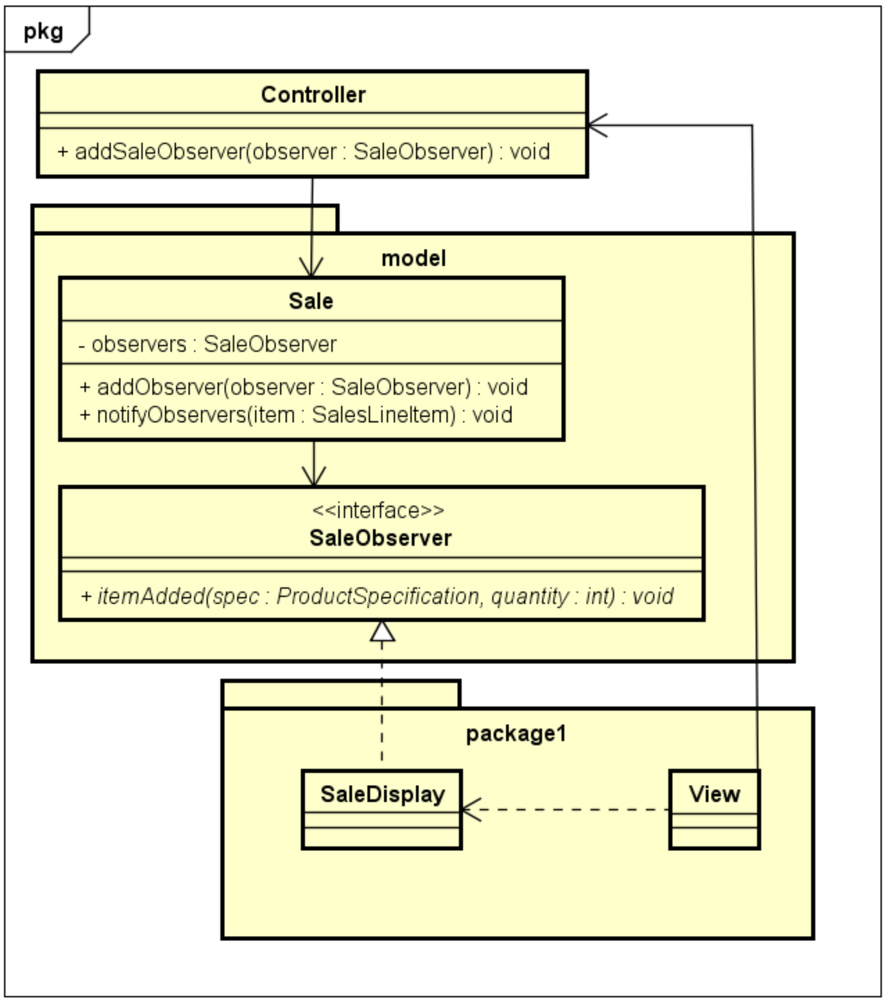

pkg
Controller
+ addSaleObserver(observer : SaleObserver) : void
model
Sale
- observers : SaleObserver
+ addObserver(observer : SaleObserver) : void
+ notifyObservers(item : SalesLineItem) : void
<<interface>>
SaleObserver
+ itemAdded(spec : ProductSpecification, quantity : int) : void
package1
SaleDisplay
View

Fig 3.4

- **Task 2b**

The implementation of the chosen design patterns where so fundamentally simple that they hardly require any explanation at all, nevertheless, here's some diagrams explaining the changes I made.

The first design pattern I decided to implement was the Singleton pattern unto the ProductCatalog class, as illustrated in fig 3.5, no changes to core functionality was made except for making sure that only one instance of this object is created using static declarations and making the constructor accessibility private.

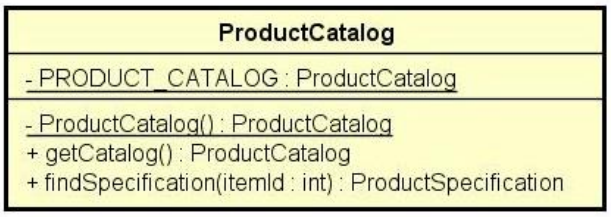

ProductCatalog
- PRODUCT_CATALOG : ProductCatalog
- ProductCatalog() : ProductCatalog
+ getCatalog() : ProductCatalog
+ findSpecification(itemId : int) : ProductSpecification

Fig 3.5

The second design pattern I decided to implement was the Iterator pattern unto the Receipt class, this time also making sure to preserve core functionality, the relevant additions are illustrated in fig 3.6.

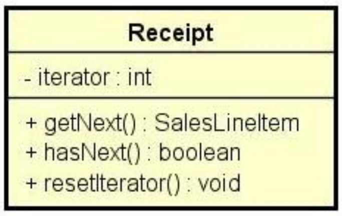


Fig 3.6

Sample run printout:

```
######################################
### List of items in current sale ###
### SALE ITEM #1: Total cost: 10 Quantity: 1 Product id: 1, name: low fat milk, price:10
description: a very long description, a very long description, a very long description
######################################

17/05/17 17:16> 1 amount of following item added to sale: Product id: 1, name: low fat milk, price:10
description: a very long description, a very long description, a very long description

######################################
### List of items in current sale ###
### SALE ITEM #1: Total cost: 10 Quantity: 1 Product id: 1, name: low fat milk, price:10
description: a very long description, a very long description, a very long description
### SALE ITEM #2: Total cost: 10 Quantity: 1 Product id: 2, name: butter, price:10
description: a very long description, a very long description, a very long description
######################################

17/05/17 17:16> 1 amount of following item added to sale: Product id: 2, name: butter, price:10
description: a very long description, a very long description, a very long description


######################################
### List of items in current sale ###
### SALE ITEM #1: Total cost: 10 Quantity: 1 Product id: 1, name: low fat milk, price:10
description: a very long description, a very long description, a very long description
### SALE ITEM #2: Total cost: 10 Quantity: 1 Product id: 2, name: butter, price:10
description: a very long description, a very long description, a very long description
### SALE ITEM #3: Total cost: 10 Quantity: 1 Product id: 3, name: bread, price:10
description: a very long description, a very long description, a very long description
######################################

17/05/17 17:16> 1 amount of following item added to sale: Product id: 3, name: bread, price:10
description: a very long description, a very long description, a very long description

17/05/17 17:16> Operation failed, no product with id #10 in catalog

######################################
### List of items in current sale ###
### SALE ITEM #1: Total cost: 10 Quantity: 1 Product id: 1, name: low fat milk, price:10
description: a very long description, a very long description, a very long description
### SALE ITEM #2: Total cost: 10 Quantity: 1 Product id: 2, name: butter, price:10
description: a very long description, a very long description, a very long description
### SALE ITEM #3: Total cost: 20 Quantity: 2 Product id: 3, name: bread, price:10
description: a very long description, a very long description, a very long description
######################################

17/05/17 17:16> 1 amount of following item added to sale: Product id: 3, name: bread, price:10
description: a very long description, a very long description, a very long description
```

```
17/05/17 17:16> Receipt for purchase of following items:
Product id: 1, name: low fat milk, price:10
description: a very long description, a very long description, a very long description
Product id: 2, name: butter, price:10
description: a very long description, a very long description, a very long description
Product id: 3, name: bread, price:10
description: a very long description, a very long description, a very long description
Product id: 3, name: bread, price:10
description: a very long description, a very long description, a very long description

Total purchase amount: 40
Amount payed: 50
Change: 10


Process finished with exit code 0
```

4 Discussion

While the presentation of the given seminar tasks might be a bit lacking, I would consider the actual implementation of the tasks to fulfill most of the assessment criteria, barring perhaps task 2b where the reasoning behind the usage of aforementioned design patterns is not very satisfactory. This is largely due to the odd nature of task 2b, normally design patterns are used to solve problems encountered when developing or designing software, not needlessly inserted into already written code.

Even when disregarding task 2b I have to say that I find it hard to motivate the usage of many GoF design patterns as many of these are decades old as this point and were often developed because of inadequacies in the C++ language at the time and are largely made obsolete by the usage of more elegant and powerful abstractions.