# Seminar 3

**Completed Version**
**Object-Oriented Design, IV1350**

Ayda Ghalkhanbaz
aydag@kth.se

May 8, 2024

# Contents

# 1 Introduction

The third task of the project involves bringing the design of the point of sale in the previous seminars into code implementation in Java.

This report covers the implementation details of classes, methods and tests of them. The code is written in Java as a Maven project using Visual Studio Code. The results then will be shown and discussed in the later sections. The link to the GitHub repository can be found on the right upper corner of the report. I cooperated with Naveed Rahman and Nora Say to accomplish this task.

# 2 Method

The first step for the tasks involved watching all the provided Video lectures and reading through the book "Object Oriented Programming". A new java project was created using Java Maven Project. The more details about task 1 and 2 are explained below.

## 2.1 Task 1

Implementing this task involves following the guidelines mentioned in chapter 6 of the book. The first step was to create a Java package and add MVC layers to the main directory of the project. For each layer, corresponding classes such as `Sale`, `SaleDTO`, `Controller`, `ItemDTO`, `ItemsInBag`, `Receipt`, `ReceiptDTO`, `View`, `StartUp` etc. were created and added to corresponding folders in each layer.

To avoid the smelly code, the methods are kept as short and simple as possible. Also, the duplicated codes have their own function and are used in different methods within the class. Additionally, data transfer objects (DTO) were passed through layers to both avoid long parameter lists and keep the program "object-oriented" which does not work only with primitive data. At the end, in order to simulate a simple interaction between each layer, hard-coded method calls were used.

## 2.2 Task 2

In the second task, each class and its methods have been tested. The tests are written using JUnit 5 library. The annotations`@BeforeEach`, `@AfterEach` and `@Test` were used to define the functionality of each test method. The first two annotations are performed before and after each test method and the last annotation defines the actual method to run for testing. The mainly used methods for validating the program are `assertEquals`, `assertTrue` and `assertFalse` to check whether the expected value corresponds to the actual value from the methods.

# 3 Result

The updated 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 Applied Changes After Completion

The applied changes include some modifications in the `View` and `Controller` classes. The `Controller` class now doesn't return any object of `ItemsInBag`, instead it is `void` now, since the `View` doesn't need to keep track of the `ItemsInBag` directly.

Additionally the `requestDiscount` method in the `Controller` class now takes in only one parameter (`CustomerID`), this is becuase the `View` doesn't interact with the model layer directly anymore so it cannot send the `saleInfo` (which is an array of `ItemsInBag`) to get a discount. Also, the `Controller` class has a new `private` method called `getFinalBag` which returns the final `ItemsInBag` array to avoid redundancy in the code since fetching the final bag is done in multiple methods in `Controller`.

## 3.2 Task 1

The program implemented for task 1 reflects a point of sale system. In this system, the cashier is able to scan items, calculate total price, discount and amount for the customer to pay, receive the payment from customer and print a receipt.

As mentioned earlier, the program is divided into layers. The controller acts as an intermediary between the view and the model layers. The model layer handles the logic and data structures of the entire program, while the integration layer takes care of storing the information and data such as accounting system, inventory system, printer etc.

On the other hand, the view class (in the view layer) creates an object of the controller and calls its methods after its needs (starting sale, scanning item, ending sale,...). The view class also simulates a fake sale to approve the functionality of the program, which is illustrated in output 3.1.

## 3.3 Task 2

The implemented tests for the controller, model and integration layers approve the functionality of the methods in the classes of these layers. The tests written for the controller layer checks the validity of the methods in the controller class. For instance,

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

Starting a new sale:

1x Item 1 added.
3x Item 2 added.
Sale has been ended

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

*****************************************************
                 Receipt

Date & Time of the sale: 2024-05-02 17:47:55

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

Total price(incl. VAT): 83,75 SEK
The amount paid: 500,00 SEK
Discount amount: 0.0
*****************************************************

Cash back to customer: 416.25 SEK

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

Figure 3.1: The output of the fake system simulation

the `registerItemTest()` method begins with creating a new shopping bag and adding items into it, then we try to add some more items with the same item ID (updating the quantity of already scanned items). The outcome of this test matches with the expected value which is the quantity `12` and item name "`Butter`". As explained in the previous chapter, the tests are applied to all relevant methods in relevant classes. All test methods can also be found on GitHub.

# 4 Discussion

The implemented program and tests for this seminar follow the Java standard conventions such as: naming conventions, indentation, spacing and documentation. Additionally all public methods, classes and packet privates have declarative comments that indicate their functionality and purposes. However commenting inside the methods has been avoided to keep the code as neat as possible.

Duplicated code is prevented since small helper methods are used during the entire program. For instance, in the Sale class, the method `containsItemID()` is used inside the `updateItemQuantityInSale()` to check whether the item ID is already scanned into the sale or not. The method will then use this information to perform its operations. Moreover the length of the classes and methods are kept as short as possible to avoid the complexity of understanding the code.

In order to keep the program in a good encapsulation, more objects are passed through layers and methods instead of just primitive data. On the other hand, `DTO`'s are used to store multiple primitive data, in this way parameter lists are also kept as short as possible.

Also, `ArrayList` is used to create a collection of the `DTO` objects. As an example, The `Sale()` class has an attribute `shoppingBag` that is an array of objects of the type `ItemsInBag`. The `ItemsInBag()` class stores information about items (`ItemDTO`) and the quantity of the purchased items. This array is then passed through layers and its attributes are used to calculate the total price or the change to pay back to the customer and at the end, after the discount process, a `SaleDTO` object is created and passed within layers that keeps additional information about the sale.

As mentioned in previous chapters, the tests are placed in the same package as SUT but in the test directory. They are written using the `JUnit` framework and all necessary annotations (`@BeforeEach`, `@AfterEach` and `@Test`) are used correctly. They also employ assert methods to evaluate the methods. Additionally, all of the test methods are self-evaluating, i.e. they don't print out any message in case of pass, but they do print an informative explanation if the test fails.

All the if-branches in the methods are tested as well. For instance, the `SaleTest` class under the test directory, evaluates both cases of `containsItemID` function sale. In other words, there are two test methods `containsItemIDTest` and `containsNoItemIDTest` that checks the validity of the if and else statements in the `containsItemID` method in the Sale class.