Seminar 3

Object-Oriented Design, IV1350

Adalet Adiljan, adalat@kth.se

4th of May 2023

Contents

1. Introduction

The purpose with seminar 3 is to start coding the basis of the program by considering the results of applying the MVC and Layers model to our user problem from seminar 2. It mainly consisted of achieving a good translation of the design model to source code and writing unit tests. Just like the previous seminars, the whole assignment was divided into two different tasks, where the first task is about writing the classes and methods to the program while task 2 was about writing tests for at least two classes to attain at least 1 point out of 2 points.

The first task required to implement the basic flow and one of the alternative flows (3-4b) according to the first seminar. It was being noted to only code in Java language to understand the solutions at the seminar, hence the coding of the program is centered to Java.
While building the tests, it is also important to note that the program achieves the principles of obtaining high cohesion, low coupling, and good encapsulation with a well-defined interface.

The report is divided into 3 parts apart from the introduction containing the groups method, which is being covered in chapter 2, the result to the assignment in chapter 3 and the discussion in reference to the provided assessment criteria for this seminar in chapter 4.

2. Method

This chapter of the report covers how we as a group reasoned when writing the program to our MVC and Layer model, along with our CCDs from seminar 2. We had early agreed and decided to meet to start working on the tasks in seminar 3 and solve it together, as we thought that the work could be done most smoothly. This way, one avoids trying to misinterpret the diagrams and the model, which increases the risk of incorrect code, which in turn costs time to get the unit tests to work. Also, by diving the assignments to each other in the group could also cost us a lot of time for extra work since the code needs to work in unity with the system in a way that increases the codes to pass the unit tests and the principles of single responsibilities met.

To increase the participation in the group, it has been both decided and promised to constantly come prepared for the meeting by for example having a clear understanding of the CCD diagrams and MVC and Layers model in a way that could prepare us to write them in a code form. Thanks to these preparation plans, I also believe that it helped us being able to understand the model and the diagrams in a more complex way, which explains the number of ideas being exchanged and applied in the code which meanwhile could be optimized at the same time.

This seminar required us to download another program called NetBeans where the functionality could provide us to create unit test codes for the corresponding codes in our program. I on the other hand, had a struggle with downloading NetBeans on my MacBook and instead had to download an extension of the NetBeans program in Visual Code. Regardless, the test was successfully implemented, and the necessary number of classes passed the tests. We were also thinking about which one of the packages between the view, integration and model was the most suitable to start building codes from since the DTO package consists of primitive data that should only be used from when necessary. Since it largely depends on the specific requirements of the application, we knew that it was suitable to start building codes from the integration package. It made sense so, since the integration layer is responsible for integrating the application with external systems that mainly deals with the storage and retrieval of data which is being depicted by the basic flow.

3. Result

This chapter of the report consists of images that are the result of each test that we have applied for respective class and methods that were considered relevant and necessary. The tests are applied with an inspiration and close consideration to the !NO criteria for applying the JUnit tests in chapter 7. The first image is a printout of our sample program when running the code.

Image 1: Printout sample of the receipt. Here we see a printout sample (receipt) of the goods purchase when the program is following a basic flow and the 3-4b alternative flow including the corresponding time, date, month, and the different VAT – rates given for each different item purchase. As the items has been purchased and the sale has ended, the inventory system is being updated hence the given message in the end is returned.

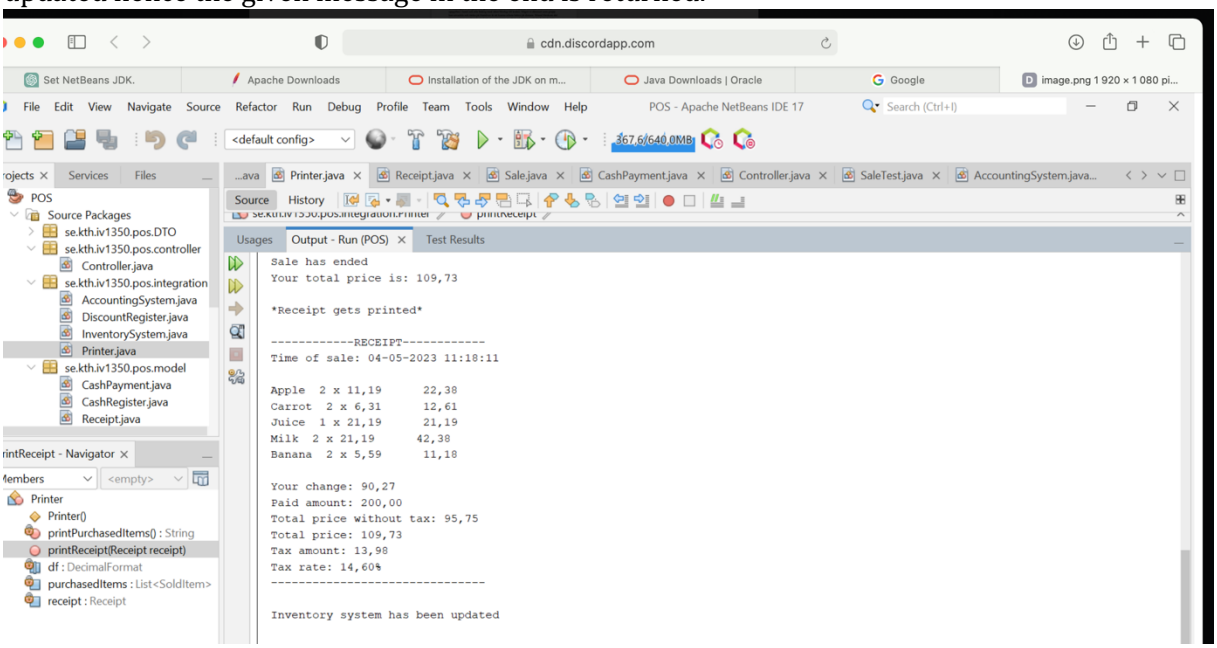


Image 2:

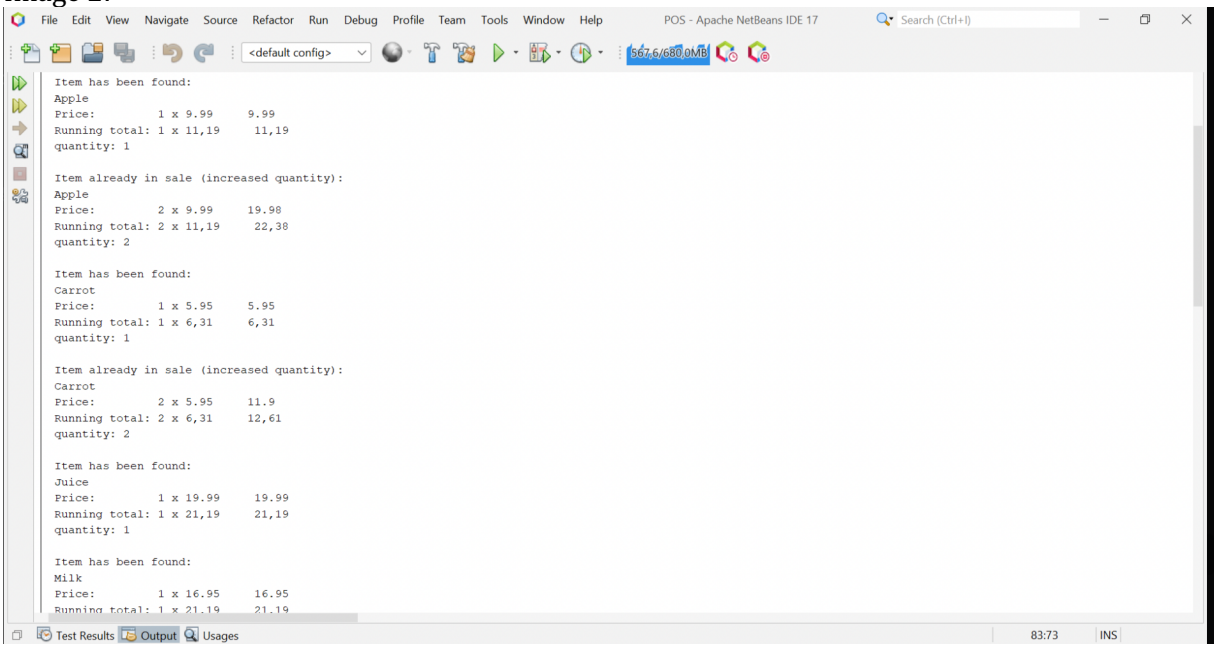


Image 3:

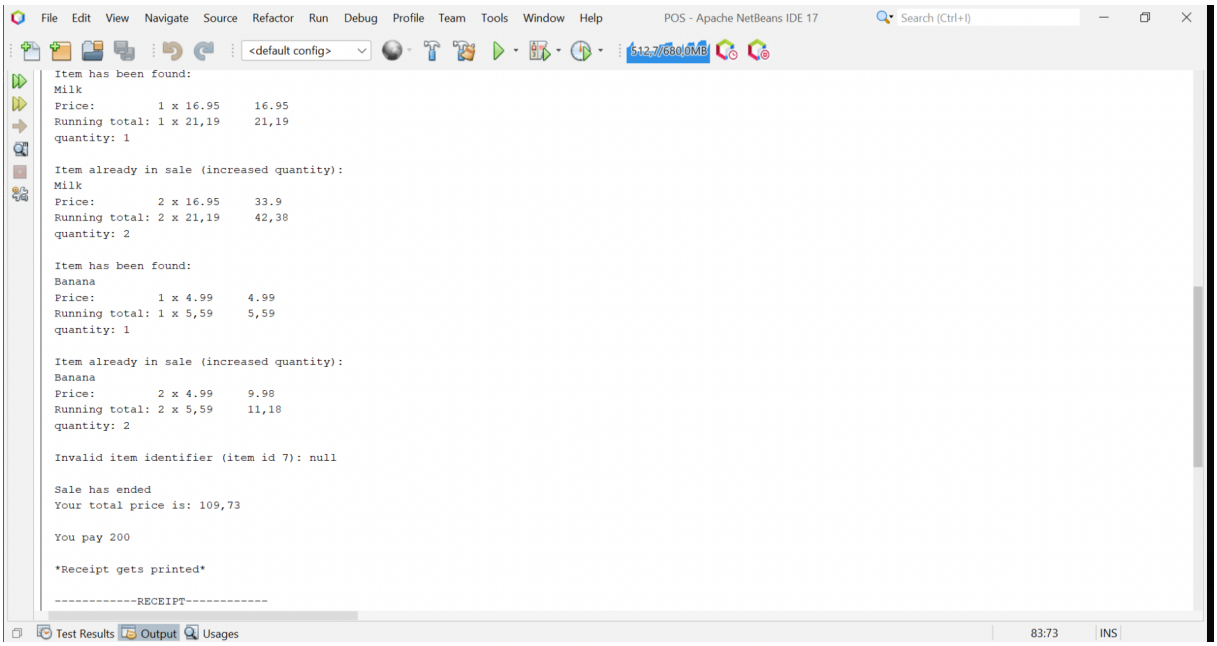
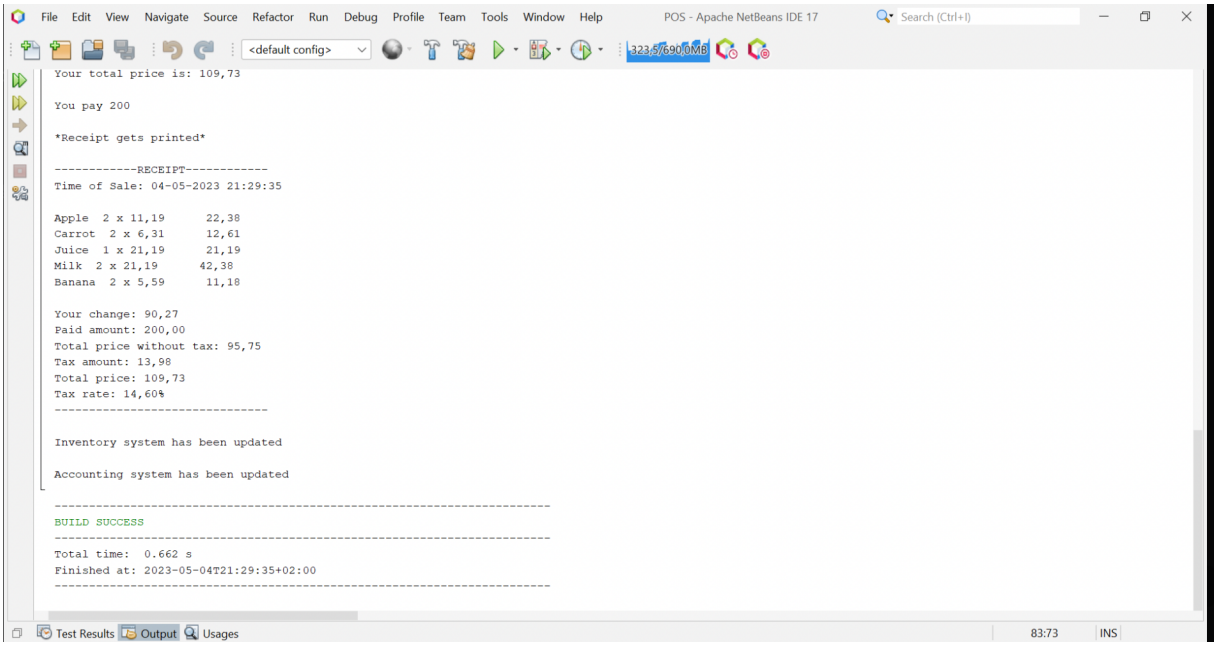


Image 4:



The images 2-4 can be explained with following brief explanation: The program keeps track of scanned items by adding them to a list of sold items in the model layer. If an item has already been scanned, its quantity is updated, while the view displays information about each scanned item. Once all items have been scanned, the view notifies the controller to end the sale. The total price with tax is calculated and presented to the view. When the customer pays, the view informs the controller of the paid amount, which creates a payment object with the total price, change, and paid amount. The program prints a receipt with sale information, logs the sale, and updates external systems.
To test the inventory system, two scenarios are tested. In the first test, if the item identifier matches an item's identifier, the item is returned, and the returned item's name should match the expected name. In the second test, if the identifier does not match any item, null should be returned.
In the sale class, a test is performed to calculate the total price with tax. Two fake items are created, and the expected result is compared with the actual result value.
The controller tests the "searchMatchingItem" method, which is used by the view to search for an item by its identifier. Three outcomes are tested: 1) an item is returned if the identifier is valid, 2) null is returned if the identifier is invalid, and 3) the quantity is increased if the item has already been added to the sale.

The full code of our implemented tests units and the coding of the program is linked here: https://github.com/Haaron1/Seminarium3_POS/tree/main/pos

4.Discussion

This chapter of the report covers the reflections by the given reflection questions in the assessment criteria for seminar 3.

The first question is about whether if the code is easily understood or not, and if all declarations have explaining names, which I would agree upon. The book has so many good examples that we have had a good use of and that gave us a clear perspective on how the methods and classes could be decelerated. An important note that we could consider as a group was how we could implement the Javadoc comments, since the way they would be formulated and the amount of detailed information in the comment could reveal a lot about if the method and the codes are contributing to the high cohesion, low coupling, and good encapsulation. With that being said, the comments were giving us clear guidelines before feeling satisfied with defining and understanding the purpose of the method or the class. Except of many clear examples from the book, we also believed that the recorded lectures on how the code was implemented for the corresponding examples of the CCD and MVC and Layers model also helped us gaining a good understanding on the importance of following the java naming conventions thoroughly. As someone is consistently using the specific naming conventions for variables, methods respective classes and packages, the purpose of the function of the code becomes clearer.

Regarding the second criteria, there was a moment where we noticed that we accidentally created a duplicated code. We noticed that since the codes appeared identical and therefore was repeated in two places within the software application. We already had a similar code in our class called itemDTO. The classes were similar since they had the same attributes as the soldItemDTO with an additional attribute that was only included for the other class. The solution to the duplicated code was clear, as one single additional attribute was not enough to keep the class soldItemDTO.

The fourth criteria follow concerning if we are following the primitive data or static members instead of objects, which my answer to that would be the objects with the reference to our previous CCD diagrams from the seminar 2. These diagrams and the MVC and Layers model has been of a good use for most of the implementation so far, since we have already been familiar to the concept of these principles. Although the model and the diagram were giving us clear guidelines, it was not always easy since our previous experiences contained mostly about handling primitive data. There are couple of things that we had to consider when making sure when to handle between primitive data and objects, that is to understand the difference between them. While the primitive data are basic data types that represent simple values such as integers, booleans etc. objects in other hand are instances of classes that can have methods and properties. Since our MVC model had a DTO package for all the primitive data, it gave us such a clear overview of the relations between the packages and their respective purposes, it has helped us significantly when it came to in what moment it was necessary to use the primitive data over an object.

Regarding the 7th criteria, I agree upon that the MVC and Layer patterns was being used correctly since we had a good use of both following the diagram and the patterns while implementing the codes so far. I also believe that the comments for the classes and the methods that could be commented both in short and in a simple way was a good indication of them being implemented as expected, not to mention the unit tests codes was being passed.

Is there one test class per test method? The answer to that question is yes. In JUnit, we have noticed that it is common to implement multiple unit tests for a single class, because of the number of functions that is included. Depending on the function, each test has to be implemented uniquely so that they had to test for each method independently. An example of a class where we had to implement two tests is the InventorySystem's findItem() method, where each test covers different possible conditions. The first test method, called testFindItemTheItemIsfound() checks if the correct item has been found by the given item ID, it will set the variable called itemId to 1, where apple is the matching item to that value (an ItemDTO). The second test executes and sets the following itemId to 6 (value is not valid) which is the finditem method returns as null.