Seminar 3

Object-Oriented Design, IV1350

Mostafa Faik

2025-05-04

Project Members:
[Mostafa Faik, mfaik@kth.se]
[Derfesh Mariush, derfesh@kth.se]
[Allan Al Saleh, Allan2@kth.se]

Declaration:

By submitting this assignment, it is hereby declared that all group members listed above have contributed to the solution. It is also declared that all project members fully understand all parts of the final solution and can explain it upon request.

It is furthermore declared that no part of the solution has been copied from any other source (except for resources presented in the Canvas page for the course IV1350), and that no part of the solution has been provided by someone not listed as a project member above.

1 Introduction

This task involves implementing a basic point-of-sale (POS) system using object-oriented principles and a layered software architecture. The system follows the Model-View-Controller (MVC) and Layered architectural patterns to ensure separation of concerns, maintainability, and testability. The implementation builds upon the design from the previous task and includes key system operations such as initiating a sale, registering items, displaying totals, processing payments, and interacting with external systems like inventory and accounting. The code is organized into well-defined packages corresponding to each architectural layer, demonstrating how design concepts are translated into a working software solution.

2 Method

The implementation of the POS system followed a structured and iterative approach guided by the design created in the previous task. The focus was on adhering to the principles of object-oriented design, such as high cohesion, low coupling, and clear separation of concerns, while applying the Model-View-Controller (MVC) and Layered architectural patterns. The steps used in this seminar is based on principles from the course literature "A First Course in Object-Oriented Development" by Leif Lindbäck, and guided by examples from lectures and exercises.

To begin, the application's architecture was divided into distinct layers—Startup, View, Controller, Model, Integration, and DTO—to clearly separate responsibilities. This helped maintain modularity and made the system easier to understand, extend, and test.

The Controller layer was implemented first, as it coordinates the overall flow between the View and the Model. The controller methods mirror key use-case steps: initiating a sale, registering items, finalizing the sale, and processing payments. Each method was designed to be concise and delegate specific responsibilities to the Model or Integration layers.

Next, the View layer was created to simulate a cashier's interaction with the system. It calls the controller methods in sequence to mimic a typical sale. Print statements were used to display item information, running totals, and final outputs, making it easy to verify system behavior during development.

The Startup layer consists of a Main class responsible for bootstrapping the application. It instantiates the Controller and passes it to the View, thereby initiating the program flow.

Throughout development, the system was tested incrementally after each implementation step. This allowed for early detection of logic errors and ensured each layer interacted correctly. Emphasis was placed on encapsulation, where each class exposes only the necessary public methods and hides internal logic. The use of Data Transfer Objects (DTOs) ensured that data could be passed safely between layers without exposing internal data structures.

When writing the unit tests for the Controller layer, we followed a systematic and functionality-driven approach to ensure that the key operations of the system were correctly implemented and behaved as expected. Our objective was to validate the controller's ability to coordinate and manage the core flow of a sale: initiating a sale, registering items, computing totals and VAT, ending the sale, and handling payment.

We began by identifying the main use cases based on the system's requirements and interaction diagrams. Each use case was translated into at least one corresponding test

case. For example, the initiateSale() method was tested to ensure it correctly sets up the internal state needed to begin a transaction. Similarly, the registerItem(String id) method was tested not only for correct item registration but also for cumulative behavior when the same item is registered multiple times.

We prioritized positive test cases that confirm the expected behavior under normal conditions:

- Registering a valid item returns a correct ItemDTO.
- Registering the same item multiple times updates the quantity and running total.
- Ending the sale returns the total price.
- Concluding the sale with sufficient payment returns the correct change.

We also tested aggregated outputs, such as the running total and VAT, to verify that the controller correctly calculates and exposes this information.

Our reasoning focused on covering the main flow first before introducing edge cases. We ensured that each controller method had at least one test verifying that its core functionality worked as intended. We also wrote assertions to check not only for non-null results but also for the correctness of values—for example, checking that totals and VAT are greater than zero when items have been registered.

Finally, we structured each test method to be independent and isolated, using @BeforeEach to reset the controller state before each test. This ensures consistent and reproducible results, which is essential for reliable unit testing.

3 Result

The final program is a console-based simulation of a Point of Sale (POS) system, implemented using the Model-View-Controller (MVC) and Layered architecture patterns. The system allows a cashier to initiate a sale, register multiple items, calculate running totals including VAT, and conclude the sale by processing a payment and printing a receipt.

The source code is publicly available in the following GitHub repository:
https://gits-15.sys.kth.se/derfesh/IV1350-VT25-Objektorienterad-design

When executed, the system runs a simulated sale scenario defined in the View class. The simulation includes registering several items—some multiple times—ending the sale, and paying with a 500 SEK bill. The output is printed to the console at each step to reflect the current state of the sale.

Below is a sample output from a typical program run:

```
Item added:
Item ID: 1
Item name: BigWheel Oatmeal
Item cost: 29.90 SEK
VAT: 6.0%
Item description: BigWheel Oatmeal 500 ml

Total cost (incl VAT): 31.69 SEK
Total VAT: 1.79 SEK

Item added:
Item ID: 1
Item name: BigWheel Oatmeal
Item cost: 29.90 SEK
VAT: 6.0%
Item description: BigWheel Oatmeal 500 ml

Total cost (incl VAT): 63.39 SEK
Total VAT: 3.59 SEK

Item added:
Item ID: 2
Item name: YouGoGo Blueberry
Item cost: 14.90 SEK
VAT: 6.0%
Item description: YouGoGo Blueberry 240 g

Total cost (incl VAT): 79.18 SEK
Total VAT: 4.48 SEK

Item added:
Item ID: 3
Item name: Luxury Bread
Item cost: 49.90 SEK
VAT: 12.0%
Item description: Just a normal bread
```

Figure 1: Sample output of the program

```
Item added:
Item ID: 3
Item name: Luxury Bread
Item cost: 49.90 SEK
VAT: 12.0%
Item description: Just a normal bread

Total cost (incl VAT): 135.07 SEK
Total VAT: 10.47 SEK

Item added:
Item ID: 4
Item name: Golden Apple
Item cost: 149.90 SEK
VAT: 25.0%
Item description: It's a golden one

Total cost (incl VAT): 322.45 SEK
Total VAT: 47.95 SEK

Item added:
Item ID: 1
Item name: BigWheel Oatmeal
Item cost: 29.90 SEK
VAT: 6.0%
Item description: BigWheel Oatmeal 500 ml

Total cost (incl VAT): 354.14 SEK
Total VAT: 49.74 SEK

End sale:
Total cost (incl VAT): 354.14
Accounting system updated with payment: 500.00
Inventory system updated
```

Figure 2: Sample output of the program

```
----------------------- Begin receipt -----------------------
Time of Sale: 2025-05-01 18:53

BigWheel Oatmeal 3 x 29.90 89.70 SEK
YouGoGo Blueberry 1 x 14.90 14.90 SEK
Luxury Bread 1 x 49.90 49.90 SEK
Golden Apple 1 x 149.90 149.90 SEK

Total: 354.14
VAT: 49.74

Cash: 500.00
Change: 145.86
----------------------- End receipt -----------------------

Change to give the customer: 145.86
```

Figure 3: The receipt output of the program

This sample run demonstrates a complete transaction in the POS system. The sale begins with the cashier scanning items, starting with "BigWheel Oatmeal," which is added three times throughout the sale. Each added item displays its details (ID, name, price, VAT, and description) along with updated total cost and VAT.

The total purchase includes a variety of items:

- 3x BigWheel Oatmeal
- 1x YouGoGo Blueberry
- 1x Luxury Bread
- 1x Golden Apple

The cumulative total including VAT reaches 354.14 SEK, with a total VAT of 49.74 SEK. Upon ending the sale, the system updates the accounting and inventory systems, simulates receipt printing, and calculates the change based on a 500.00 SEK cash payment. The customer receives 145.86 SEK in change.

The receipt clearly shows each item, quantity, unit price, and subtotal, along with the total cost, total VAT, and change due. The sale timestamp is also printed for record-keeping. This output confirms that the system correctly handles item registration, VAT calculation, system updates, and receipt formatting.

The unit tests for the different layers (Model, Controller, and Integration) were conducted to ensure the proper functioning and adherence to expected behavior of the system. Below is a brief explanation of the tests for each layer:

Controller Layer Tests:

In the controller layer we wrote tests that checks the initiateSale() method to verify that initiating a sale correctly initializes the system and that the running total is not null. Tests that verifies that the registerItem() method works as expected by ensuring the item is correctly registered and that the returned ItemDTO contains the expected values such as the identifier and description. Tests that ensures that the registerItem() method updates the quantity of the item and adjusts the running total appropriately when the same item is registered multiple times. Tests that validates the endSale() method by ensuring that the total price of the items registered during the sale is correctly calculated and returned. Tests that verifies that the concludeSale() method calculates and returns the correct change when a payment is made. It ensures the change is non-null and non-negative. Tests that checks that the getRunningTotal() method correctly returns the running total after an item has been registered. Tests that ensures that the getRunningVAT() method correctly computes the VAT amount after an item has been registered.

Integration Layer Tests:

In this layer we tested and checked that initiating a sale within the controller layer properly initializes the system, ensuring the running total is set and ready for further actions. We also verified that registering an item within the system using the controller's registerItem() method functions correctly and returns the appropriate ItemDTO with the expected details such as item identifier and description. We ensured that when an item is registered more than once, the system updates the quantity correctly and reflects the change in the running total. We validated that the sale ends correctly, and the total price after all items are registered is returned as expected. We ensured that after the sale ends, the system correctly calculates and returns the change based on the given payment amount.

Model Layer Tests:

We wrote different tests that tested and ensures that the Amount class correctly initializes the amount with the provided value, validating that its internal state is set as expected. Tests that ensures that the ItemDTO constructor correctly initializes the item with its attributes, including the identifier, description, price, and VAT rate. Tests that verifies the equals() method of the ItemDTO class, ensuring that two ItemDTO objects with identical attributes are considered equal. Tests that checks that the addQuantity() method in the Item class properly increases the quantity of the item by the specified amount.

4 Discussion

The program adheres well to the software design principles and guidelines introduced during the course. Below is an evaluation of the implementation based on the provided assessment criteria:

Code Readability and Naming Conventions:

The code is overall easy to read and follow. Class, method, and variable names are descriptive and reflect their responsibilities. Java naming conventions are consistently followed throughout the codebase, which improves readability and maintainability.

Code Duplication:

Code duplication has been minimized by using clear abstractions and helper methods. For instance, logic related to printing item information, calculating totals, and formatting output is encapsulated in separate methods or classes. The only code that is considered duplicated, are some methods in Item.java and ItemDTO.java, but this was discussed and asked about during the tutorial and it was pointed out that we shouldn't be worried about this since the methods have their own and seperate use in later code files.

Method and Class Length:

Most methods and classes are kept concise and focused on a single responsibility. The system benefits from the layered and MVC architecture, which divides responsibilities into the appropriate components. No class or method is excessively long or difficult to understand (except maybe Sale.java).

Use of Objects Over Primitives:

The system makes good use of objects instead of relying on primitives or static members. For example, sale data is encapsulated in SaleDTO and item details are handled by Item objects. This approach enhances flexibility and reusability.

Parameter Lists:

Method parameter lists are generally short and appropriate. When multiple data values need to be passed, they are grouped into objects (like DTOs) to avoid unnecessarily long lists and to improve method clarity.

Commenting Practices:

All public declarations are documented using comments. The code avoids unnecessary inline comments within methods, favoring clean, self-explanatory logic. This aligns well with the guideline of having one comment per public declaration and none inside methods.

MVC and Layer Patterns:

The program structure follows the MVC and Layer patterns as intended. The View handles user interaction, the Controller manages logic flow, and the Model holds business data and logic. There is also a clear Integration layer for external system simulation and a DTO package for transferring data between layers. This architecture enforces separation of concerns and modularity.

Coupling, Cohesion, and Encapsulation:

The design exhibits low coupling and high cohesion. Each class has a clear responsibility and communicates with others through well-defined interfaces or DTOs. Encapsulation is well maintained by using private fields with appropriate getters or constructors, ensuring data integrity and hiding implementation details.

Use of JUnit Framework:

JUnit was used for all the unit tests. The test classes are based on the JUnit 5 framework, as evidenced by the use of annotations like @Test and @BeforeEach, which are part of the JUnit library. This ensures that the tests are structured, maintainable, and can be easily executed with a standard testing tool.

Self-Evaluating Tests:

All tests are self-evaluating. They use JUnit's built-in assertion methods like assertEquals, assertNotNull, assertTrue, and assertFalse. These assertions will print informative messages if the test fails, which is crucial for diagnosing problems in the code.

Test Class Organization:

There is one test class for each of the major components tested. For example, the ControllerTest class tests the Controller class, and methods in the ItemTest class test various methods of the Item class.

Coverage of Branches and Parameter Values:

The tests pass a variety of relevant parameters to the tested methods. For instance, the testConcludeSale() method passes a payment amount, and the testRegisterItemIncreasesQuantity() method checks that registering an item multiple times correctly adjusts the total.

Completeness of Tests:

The tests cover all major public methods in the classes, especially those with critical business logic. Methods like registerItem, getRunningTotal, and concludeSale are tested

with relevant assertions to ensure correct behavior. However, methods that may be simple getters or setters (such as direct access to fields in certain classes) are typically not tested unless they contain additional logic. This is a standard practice in unit testing where only methods with logic or side effects are thoroughly tested.

In conclusion, the implementation adheres to both the technical requirements of the task and industry best practices. The use of JUnit for unit testing ensures that the system behaves as expected, and the design principles contribute to a robust, maintainable, and extensible system.