Seminar 2

Object-Oriented Design, IV1350

Mostafa Faik

2025-04-07

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 report presents the design of a program that supports all flows—both main and alternative—of the Process Sale scenario, including system startup. The design follows object-oriented principles with high cohesion, low coupling, and strong encapsulation. To ensure a maintainable and scalable architecture, the solution is structured using the Model-View-Controller (MVC) and Layer patterns. The focus is placed on the logic and control layers, while the View and data storage components are simplified or abstracted. This structured approach enables clear separation of responsibilities and promotes system flexibility and testability.

2 Method

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. The design process followed a practical application of the MVC and Layer architectural patterns.

The first step was to separate the program responsibilities into layers: the View handles user interactions, the Controller coordinates system behavior, and the Model contains all domain-specific logic (such as Sale, Discount, and Receipt). This ensured that the solution achieved high cohesion and low coupling from the start.

System operations were then identified from the SSD diagram created in Seminar 1. Each system operation (e.g., initiateSale, registerItem, discount, endSale, concludeSale) was designed one at a time. For each, we developed a corresponding UML communication diagram to visualize message flow and clarify how responsibilities should be distributed between objects.

The communication diagrams were designed in the logical execution order. For example, initiateSale() was designed before registerItem() since the sale must be initialized before any items are added. These diagrams guided implementation by clearly showing how methods interact, what parameters are passed, and what values are returned.

Design choices prioritized key object-oriented principles. For example, endSale() was modeled as a separate operation to keep the responsibility of computing the total price (including discounts) within the Sale class.

Throughout the design process, we revisited and refined diagrams to ensure they followed principles of low coupling, high cohesion, and good encapsulation. Clear separation of concerns and single-responsibility design led to a clean and maintainable architecture.

3 Result

This section presents and explains the interaction diagrams and class diagram designed for the "Process Sale" scenario. Each interaction diagram focuses on one specific part of the system functionality, and together they describe the entire process from startup to concluding a sale. All diagrams have been created using the principles of the MVC and Layer patterns to ensure high cohesion, low coupling, and proper encapsulation.

3.1 Startup diagram

This diagram shows how the application initializes when the program starts. It demonstrates the setup of the core components and how they are interconnected. The purpose is to show how the main method is responsible for creating the system and its dependencies, following proper layering and separation of concerns.

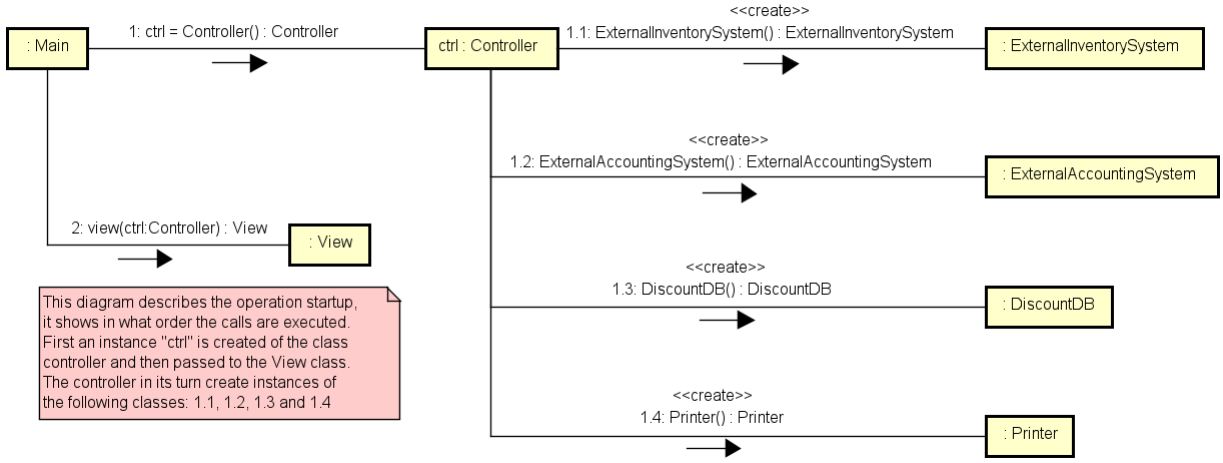

Figure 1: Startup Communication Diagram

The main method first creates an instance "ctrl" of the class controller and then passed to the View class. The controller in its turn creates instances of external systems like Printer and ExternalInventorySystem. The Controller is the central hub that coordinates the flow between the view and the model layer. After instantiating the Controller, the main method creates a View object, which simulates user interaction. This design respects low coupling by injecting dependencies rather than hardcoding them and maintains high cohesion by letting each class manage only its own responsibilities.

3.2 StartTheSale diagram

The diagram below represents the sequence of interactions that take place when a cashier initiates a new sale in the system.

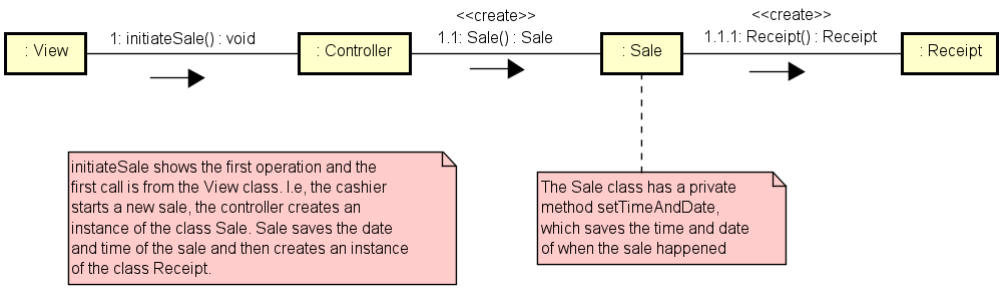

Figure 2: StartTheSale Communication Diagram

This diagram shows the first step in the Process Sale scenario, where the cashier starts a new sale. The View calls initiateSale() on the Controller, which then creates a new Sale object. When the Sale object is created, it saves the current date and time using a private method. It also creates a Receipt object that will be updated throughout the sale. This interaction sets up the system to begin registering items and ensures each sale has a timestamp and a corresponding receipt.

3.3 RegisterItem diagram

This interaction diagram illustrates how the system handles item registration. It includes both the basic flow where items are entered one by one and alternative flows where quantity can be specified or existing items are updated.

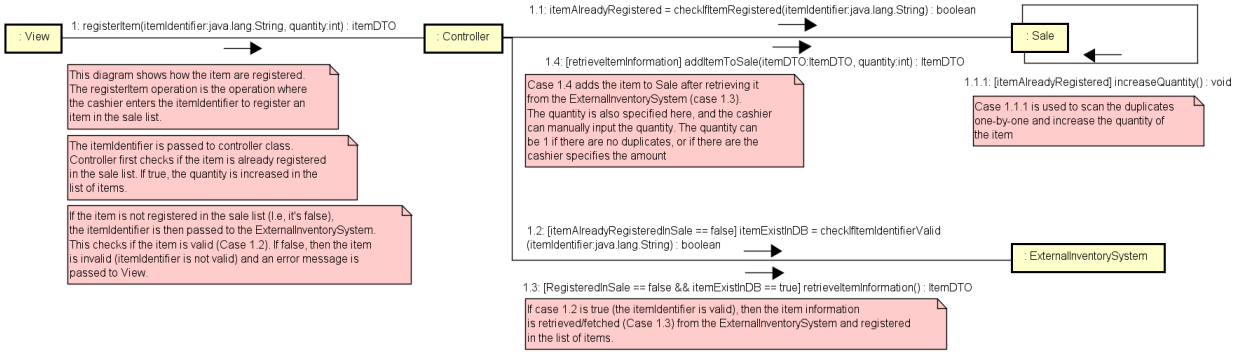

Figure 3: RegisterItem Communication Diagram

When the cashier scans an item or enters its identifier, the View calls registerItem on the Controller. The Controller forwards this request to the Sale object, which checks if the item already exists in the current sale. If the item is already registered, the quantity is updated instead of creating a new entry (case 1.1.1 is used for scanning the duplicated

items one-by-one), as per the SSD. This behavior ensures data consistency and efficiency. If the item is not registered, then the controller forwards the request to case 1.2 to check if the item is valid or not. If the item is valid, then case 1.3 is used to retrieve the itemInformation from the ExternalInventorySystem and then registered in Sale in case 1.4. In case 1.4 the cashier can specify the quantity of the item (manually). If there are no duplicates of the item, then the cashier can input 1 as the quantity (otherwise, the cashier specifies the amount and inserts it as the quantity).

3.4 Discount diagram

This diagram represents the optional step where a discount is applied to the ongoing sale based on customer ID and predefined discount rules.

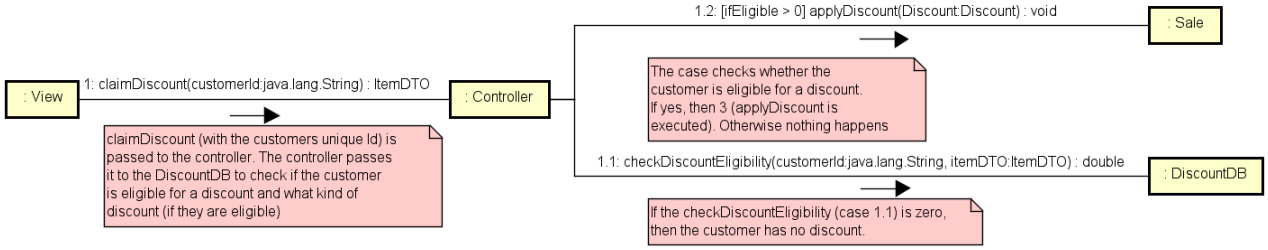

Figure 4: Discount Communication Diagram

When the cashier enters a customer ID, the View sends this information to the Controller, which then calls checkDiscountEligibilty to check if the customer has a discount and what type of discount they have. If the customer has a discount then the method applyDiscount(customerID) on the Sale is called. If the customer does not have a discount, then nothing happens and the applyDiscount method is not called. The Sale ensures that discount application logic is encapsulated and remains within the domain layer, keeping the Controller free of any business logic.

3.5 EndSale diagram

This diagram describes the operation that ends the item registration process and calculates the total price, including any applicable VAT and discounts. It happens after the cashier confirms that all items are entered.

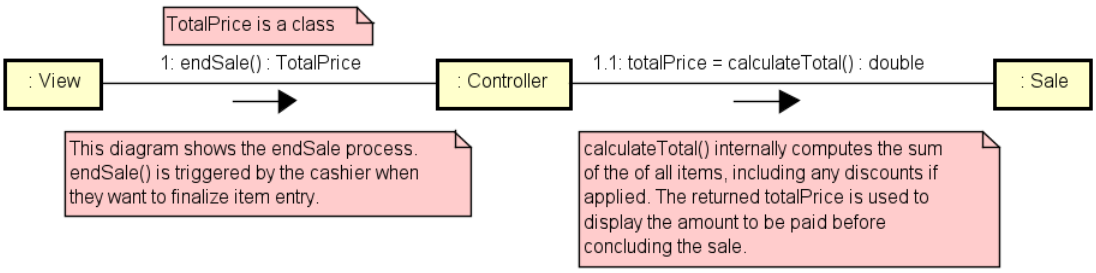

Figure 5: EndSale Communication Diagram

The View calls the endSale() method in the Controller. This triggers the Controller to call calculateTotal() in the Sale object. The Sale then performs all final calculations, including applying VAT and summarizing the total price. The result is returned up the call stack and displayed by the View. This clean separation ensures the Controller remains a coordinator, while all calculation logic is centralized in the Sale class, ensuring high cohesion.

3.6 ConcludeTheSale diagram

This diagram shows the final step of the sale where payment is made, and a receipt is printed. It includes the creation of the receipt and interaction with external systems.

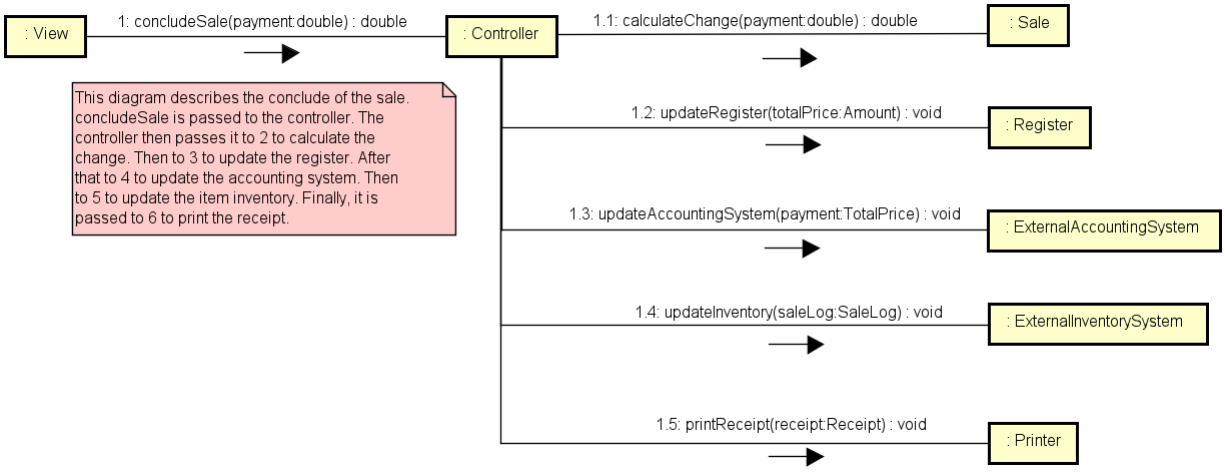

Figure 6: ConcludeTheSale Communication Diagram

The View initiates the concludeSale operation, which is passed to the Controller. The Controller then calls calculateChange in the Sale, to calculates the change. Then, updateRegister is called to update the register with the total price that is paid. After that,

the accounting system of the store is updated by calling the method updateAccountingSystem. Then,, the inventory of the store is updated in the updateInventory method. Finally, the receipt is passed to the Printer to generate a physical copy by calling the printReceipt method.

3.7 Class Diagram – Process Sale System Architecture

This class diagram represents the overall structure and architecture of the "Process Sale" system. It shows all key classes involved in the interaction diagrams and how they are distributed across layers according to the Model-View-Controller (MVC) and Layered architectural patterns. It illustrates the relationships between objects, the responsibilities of each class, and the communication pathways between them.

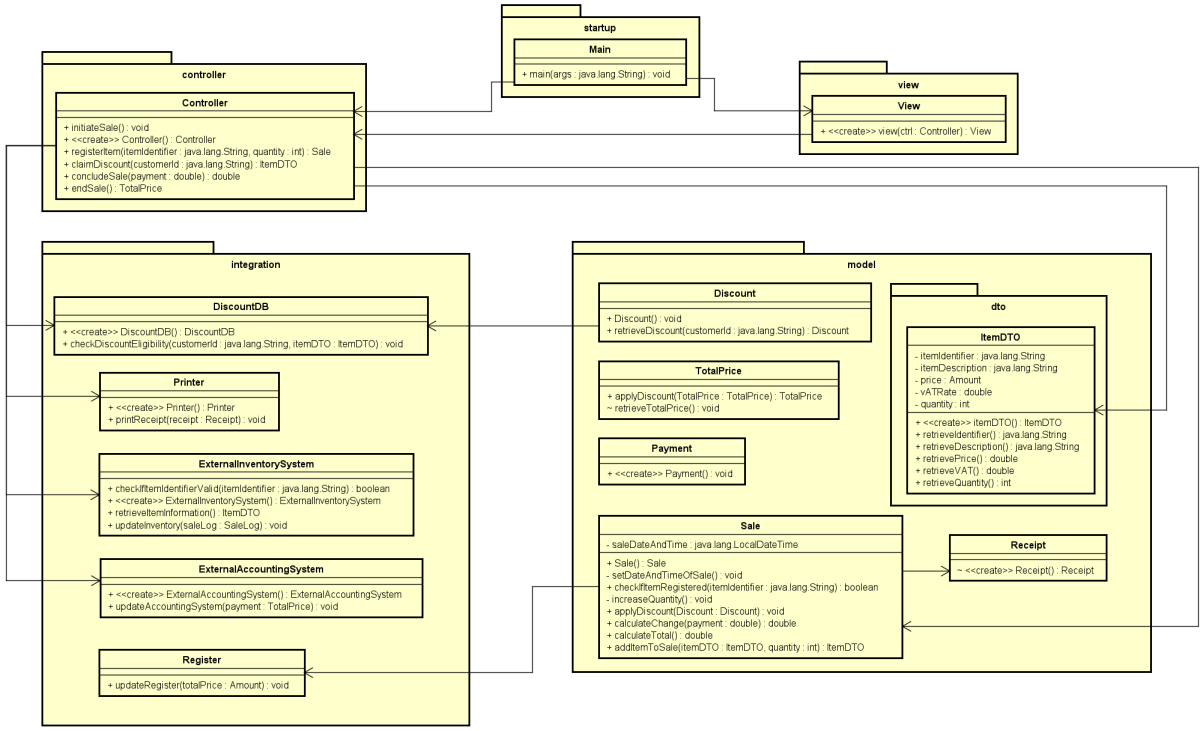

Figure 7: Process Sale System Architecture

At the top level, the Main class is responsible for starting the application. It serves as the entry point, where instances of the controller and view are created and linked. The View class acts as the user interface, simulating the cashier's interaction with the system. It does not contain any business logic; instead, it forwards user commands to the Controller.

The Controller class is central in the controller layer and coordinates all the actions

in the system. It receives inputs from the View and invokes operations in the model layer accordingly. For example, it handles starting and concluding a sale, registering items, checking discounts, and computing totals. The controller does not perform any logic directly, which helps preserve low coupling and high cohesion.

In the model layer, the Sale class is the core of the application logic. It encapsulates the responsibilities of registering items, applying discounts, calculating the total price including VAT, and handling payments. The Payment class records the amount paid, and the Receipt class represents the final sale record generated upon completion.

This design showcases strong design principles. High cohesion is maintained by ensuring each class has a single, well-defined responsibility. Coupling is kept low by ensuring that classes communicate through interfaces and that the controller mediates communication between layers. Encapsulation is also respected by hiding internal logic and exposing only necessary public methods. The layering of the system ensures that each concern—user interaction, control flow, business logic, and external integration—is modular and can be modified independently.

4 Discussion

The design and diagrams presented in this task are structured to be easy to understand, following a logical flow from user interaction through the controller to the model, which aligns with the MVC and Layer design patterns. The responsibilities are clearly separated: the View handles user interaction, the Controller mediates between the view and model without containing business logic, and the Model holds the core functionality such as handling sales, receipts and discounts.

The layering is adequate for the scope of the task. We use the standard separation into startup, view, controller, model, and integration layers. Each layer contains the appropriate number of classes for its responsibilities, avoiding unnecessary complexity. There is no mixing of responsibilities: the controller does not contain business logic, and the model contains no UI-related code.

The design follows key object-oriented principles—low coupling, high cohesion, and good encapsulation. Each class has a focused responsibility (high cohesion), classes interact through well-defined interfaces (low coupling), and internal data is protected and accessed through methods (encapsulation). These principles are discussed and demonstrated in both the Method and Result sections, where interaction and class diagrams support the design reasoning.

Parameters and return types are appropriately chosen and clearly reflect the flow of data. For example, the registerItem() method receives a string identifier and quantity, and returns an ItemDTO, which encapsulates all relevant item data in one object. This

shows that objects are preferred over primitive types to group related data and reduce method complexity.

No static methods or attributes are used inappropriately. All objects are instantiated where needed to support flexibility and maintainability. Additionally, objects do not expose their internal attributes directly; instead, they use methods like retrievePrice() or retrieveQuantity() to safely access data. This preserves object integrity.

Java naming conventions are followed throughout the design. Class names use PascalCase, method names use camelCase, and variables are named descriptively to clearly convey their purpose. This makes the codebase easier to read and maintain.

In conclusion, the design demonstrates a strong understanding and correct application of object-oriented principles and architectural patterns. The use of MVC and Layer patterns, along with high cohesion, low coupling, and encapsulation, contributes to a well-structured, maintainable, and scalable system. The diagrams are easy to read, follows Java conventions, and uses objects appropriately to model real-world concepts.