Seminar 1

Object-Oriented Design, IV1350

Mostafa Faik

2025-03-24

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 development of a domain model and a system sequence diagram (SSD) for a retail store's sales process. The purpose of this task is to design a structured representation of the system's key components and interactions, ensuring compliance with the specified requirements.

The first task involves creating a domain model that accurately represents the entities and their relationships within the retail store system.

The second task requires drawing a system sequence diagram (SSD) that illustrates the flow of interactions between the cashier and the system during a sale. The SSD must capture both the basic and alternative flows of the process, from item scanning to payment processing, discount verification, and receipt generation.

2 Method

The main sources and methods used throughout this seminar task, were the videos for the lectures and the course book "A First Course in Object Oriented Development, Leif Lindbäck".

To develop the domain model, we used category listing and noun identification to identify potential classes. The category list included key concepts related to a retail store system, such as entities (e.g., Item, Customer, Sale) and external systems (e.g., External Inventory System, External Accounting System). By analyzing the problem description, we extracted nouns that represented real-world objects and determined which ones should be modeled as classes.

Attributes were identified based on the relevant information each class needed to store. For example, the ItemInformation class required attributes such as itemIdentifier, price, and vATRate, while the Customer class included customerId for discount eligibility. Associations between classes were established by analyzing interactions.

For creating the UML diagrams, we used a UML modeling tool, ensuring clarity in visualizing relationships and dependencies within the system.

When developing the SSD, we first analyzed the basic and alternative flows from the requirement specification. We identified system events triggered by the cashier and customer interactions, such as entering an item identifier, processing payment, and applying discounts.

To ensure completeness, we followed a step-by-step approach:

- Identified external actors interacting with the system (e.g., cashier, external systems).
- Mapped interactions based on the sequence of operations in the sales process.
- Incorporated conditional flows for handling invalid items, repeated items, and discounts.
- Used a UML sequence diagram tool to illustrate interactions clearly.

By following this structured approach, we ensured that the SSD correctly reflected the system's functionality while maintaining clarity and consistency with the domain model.

3 Result

The developed domain model represents the structure of the retail store system, illustrating the key entities, their attributes, and associations. Figure 1 below presents the UML class diagram of the domain model.

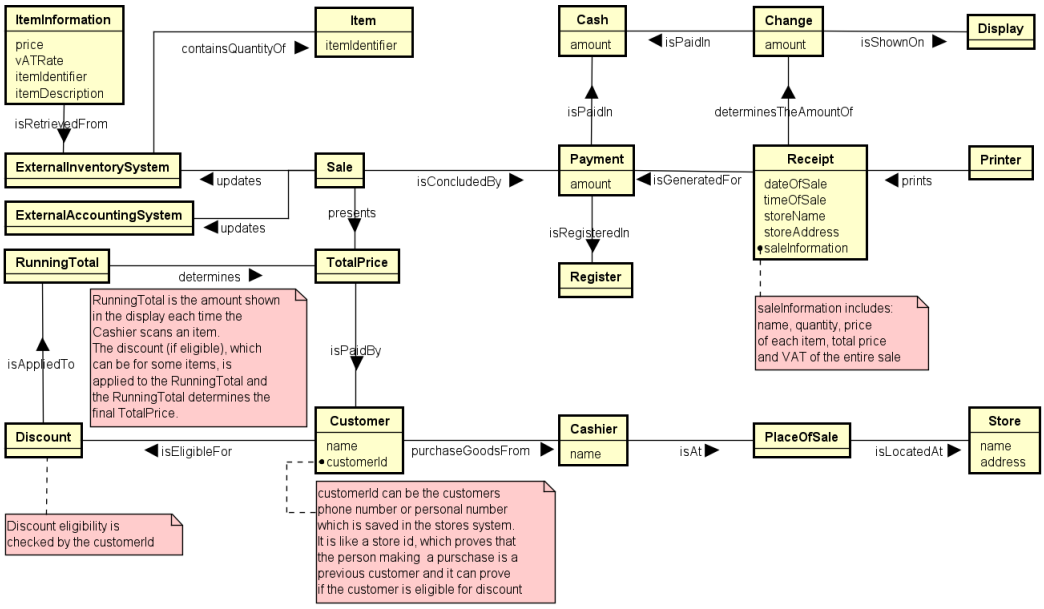

Figure 1: Retail Store Domain Model

From the model we can see that *Sale* is the core entity in the model and is responsible for updating the *ExternalAccountingSystem* and *ExternalInventorySystem*. It also presents the *TotalPrice* to the customer and is concluded by a *Payment*.

The *ExternalInventorySystem* contains *Item* quantities and provides *ItemInformation*, including price, VAT rate, item identifier, and description. The *ExternalAccountingSystem* records completed sales transactions.

Item represents a product with an itemIdentifier. *ItemInformation* stores details such as price, vATRate, itemIdentifier, and itemDescription, fetched from the *ExternalInventorySystem*. *TotalPrice* determined by the *RunningTotal*, which factors in applicable *Discounts*.

Discounts apply to the *RunningTotal* if the *Customer* is eligible (based on customerId). The *Customer* has attributes name and customerId, which determine eligibility for discounts. The *Cashier* manages the transaction and is present at a *PlaceOfSale*, which is located in a *Store* (with name and address). The *Customer* interacts with the *Cashier* to complete a purchase.

The *Payment* (amount) is made in *Cash* and registered in the *Register*. The *Receipt* includes details like dateOfSale, timeOfSale, storeName, storeAddress, and saleInformation (item details, total price, and VAT). The *Printer* prints the *Receipt*, which deter-

mines the *Change* (amount). The *Change* is paidIn *Cash* and shownOn *Display*, ensuring the customer knows how much they receive back.

The System Sequence Diagram (SSD) in Figure 2 illustrates the Process Sale use case in a retail store system. It models the interactions between the *Cashier*, *System*, and various external components such as the *Display*, *ExternalInventorySystem*, *ExternalAccountingSystem*, and *Printer*.

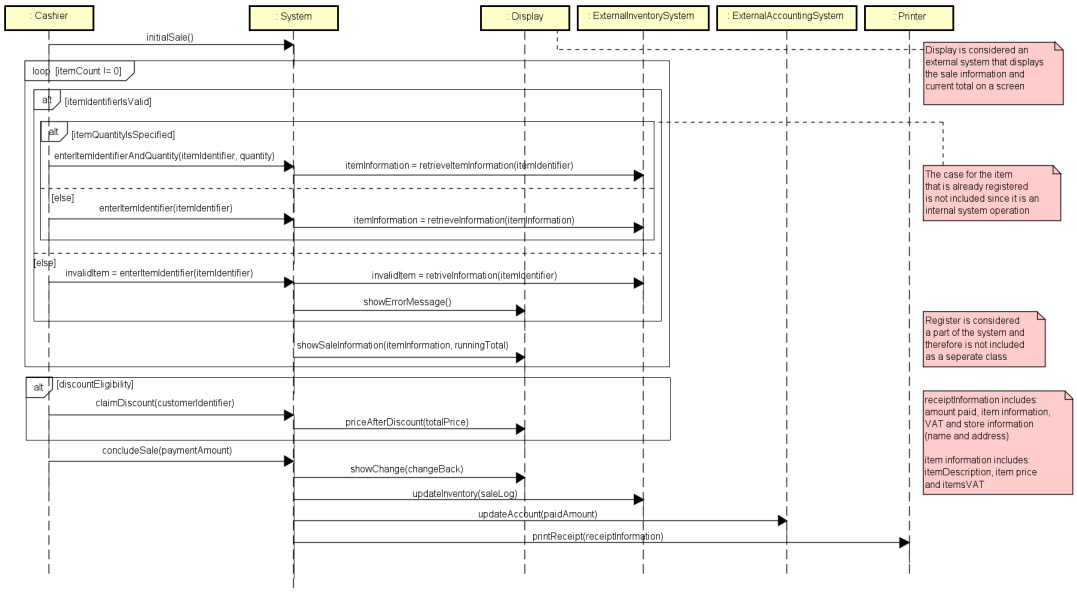

Figure 2: Retail Store SSD

The cashier starts the sale process by calling *initialSale()*. The cashier enters an item identifier, and the system validates the identifier by fetching item information from the *ExternalInventorySystem*. If the item is valid, its description, price, and VAT are retrieved and displayed on the screen *(showSaleInformation(itemInformation, runningTotal))*. If the item is invalid, an error message is displayed *(showErrorMessage())*. If the item quantity is specified, the system handles it separately.

If the customer is eligible for a discount, the cashier enters the customer identifier, and the system calculates the discounted price *(priceAfterDiscount(totalPrice))*. Once all items are entered, the cashier concludes the sale by processing the payment *(concludeSale(paymentAmount))*. The change is calculated and displayed *(showChange(changeBack))*. The system updates the inventory and accounting records *(updateInventory(saleLog), updateAccount(paidAmount))*.

A receipt is printed containing all relevant details: Amount paid, Item information (name, quantity, price, VAT) and Store details (name and address). The register is inte-

grated within the system, rather than being a separate class. Internal system operations, such as handling already registered items, are not explicitly included in the diagram.

This SSD effectively captures the main sequence of events in a retail sale, ensuring a clear and structured representation of the Process Sale use case. It models interactions accurately and concisely, ensuring correctness and usability.

4 Discussion

The domain model (DM) was evaluated based on the assessment criteria to ensure it accurately represents the Process Sale use case without being too programmatic or naïve. Below is a detailed evaluation:

Avoiding a "Programmatic" or "Naïve" DM

- The DM does not include unnecessary system-related entities such as Program or System. Instead, it models real-world entities relevant to a retail store, such as Sale, Customer, Item, Cashier, and Payment.
- A naïve DM would have lacked associations or meaningful structure, but this DM captures the relationships between classes effectively.

Clarity and Correctness

- The model is understandable and follows logical business rules. Each class has clear responsibilities, such as Sale managing transactions and ExternalInventorySystem handling stock updates.
- The Receipt correctly encapsulates all relevant sale details (date, time, store name, item details, total price, and VAT), demonstrating a correct understanding of retail transactions

Number of Classes and Missing or Irrelevant Classes

- The DM contains a reasonable number of classes, covering key aspects of a retail store.
- No essential entities are missing. Each class is justified based on the problem requirements.
- There are no irrelevant classes. Every class contributes to the retail sale process.

Spider-in-the-Web Classes

- The Sale class could be considered a spider-in-the-web as it has multiple associations. However, this is not the case and is justified because Sale is the central

entity in the Process Sale use case and this class is not considered a problem since it only represents essential associations without making the *Domain Model* unclear or hard to read.

- Other entities, such as Item, Customer, and Cashier, remain well-balanced, preventing an overly complex structure.

Choice Between Class and Attribute

- The decision to model discounts, total price, and payment as separate classes rather than attributes is justified. These entities have complex relationships and responsibilities (e.g., discount calculations and payment processing).
- ItemInformation is modeled as a separate class to encapsulate price, VAT, item identifier, and description rather than storing them as attributes in multiple classes.

Associations

- The associations are well-defined, ensuring correct interactions between entities.
- All important relationships are included, such as Sale updating both ExternalAccountingSystem and ExternalInventorySystem, and Receipt linking to Payment.
- No class is left without an association unless it is naturally independent (e.g., Printer, which only prints receipts).

Naming Conventions and UML Standards

- Consistent and clear naming conventions are followed throughout the DM. Class names are singular and describe entities accurately (Sale, Customer, Cashier).
- UML notation is used correctly, with proper associations and multiplicities where necessary.

Below is a detailed evaluation of the System Sequence Diagram (SSD)

Correct Objects and Operations

- The SSD includes all required actors and system components, such as Customer, Cashier, and POS System.
- Each operation in the Basic Flow and Alternative Flows is correctly represented, ensuring completeness.

Operations and Return Values

- The SSD includes the right level of detail, such as enterItemIdentifier and retrieveItemInformation.
- The system properly responds to invalid item identifiers, duplicate items, and discount requests, demonstrating a clear understanding of alternative flows.

Use of UML Notation

- The SSD follows correct UML syntax, with lifelines, messages, and interactions properly represented.
- The sequence of interactions follows the specified use case flows, ensuring clarity.

Overall Clarity and Completeness

- The SSD provides a clear visualization of how the Process Sale scenario functions.
- No unnecessary elements are included, and all required steps are mapped effectively.

The domain model and SSD effectively capture the retail store system's functionality. The DM avoids common mistakes, maintains clarity, and includes appropriate classes and associations. The SSD correctly sequences interactions and adheres to UML standards. Overall, the model and diagrams provide a strong and well-structured representation of the Process Sale use case.