Seminar 2

Object-Oriented Design, IV1350

Johanna Schubert jschub@kth.se

2024-04-17

Contents

1 Introduction

This report presents the designing of the Process Sale program, as outlined in the document from seminar one. I worked alone on this assignment. The objective of this project is to conceptualize a robust solution capable of managing all aspects, including alternative flows, of the Process Sale scenario, without the inclusion of actual code.
The primary focus is to ensure high cohesion, low coupling, and effective encapsulation, thereby fostering a well-defined public interface. To achieve this, the solution will adhere to established architectural patterns, notably the Model-View-Controller (MVC) and Layer patterns.
While the initial domain model and system sequence diagram created for seminar one provide a foundation, modifications have been made to rectify any identified flaws.
It is important to note that the design of the view component will be abstracted, with a placeholder class named View, and the data layer may be omitted from the design process. The focus is on design rather than implementation.
The resulting design will provide a solid foundation for the development of a robust and scalable solution to address the Process Sale scenario.

2 Method

The fist part was to divide the systems into the layers.
View Handles user interfaces, user input, and presentation logic (This was not something we had to do for this assignment.)
Controller Contains business logic, orchestrates interactions between components.
Model Represents core business entities, data, and rules.
Integration Deals with databases, external services, and communication. Such as the external accounting system, external inventory system, discount system and the printer.(We will not be designing these systems they will just represent the systems)
Startup Contains the main class an starts up the entire program.

After this we create the communication diagrams for the different operations. I divided the them up into six communication diagrams and one class diagram for the overview. When designing the diagrams i used the of Low coupling, High Cohesion and Encapsulation. Low coupling means minimizing dependencies between components. High cohesion means grouping related functionality together. Within each layer, group related components or classes. Create a separate class if it does not make sense to have some modules or attributes in the same class. Ensure each module has a clear purpose. Encapsulation Making sure to keep the public interface as small as possible. Always asking the question should this be public or private when creating operations and attributes.

3 Result

Start Sale

The first communication diagram is the start sale. So the cashier interacts with the view that is fronted part of the program which we have not designed. Here we send a operation from the view to the controller to initiate the sale. The controller then makes a new instance of the class sale, witch in turn creates an instance of Receipt. The controller also creates an instance of the Customer payment class. The Classes Recipt, CustomerPayment, Sale all have a module with the same name as the class that is public. The time that the sale starts is also saved in sale. See figure: 3.1

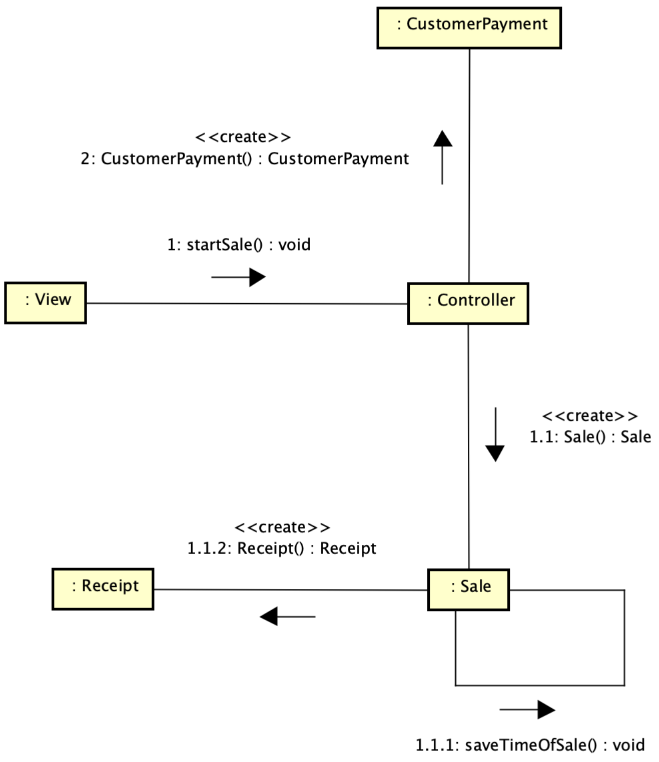

Figure 3.1: The communication diagram for startSale

Scanning Goods

The second diagram is the one that illustrates the scanning of the goods. From the view we get a string that represent the individual items. This is the bar code that the cashier scans. We also get a quantity of the goods. The controller sends an operation to the sale to check in the sale if the item is already registered. This is then added as a boolean variable. If the variable itemAlreadyInRegisterdSale is True then we can increase the quantity of the good. The variable will then be changed to False again since we have added the item to the sale. The controller can then do operation 1.2. The guard for this is that (itemAlreadyInRegisterdSale == False AND ItemExistsInDB ==True) so if this is the case we will fetch the item information from the external inventory system and put it in a ItemDTO object. The increaseQuantity() operation is a private module in the class Sale since it will not have to be accessed anywhere outside sale. The operation getItemById is in the inventory system class to be able to take the ItemID (type: String) and search it in the database. This will be looped until there are no goods left. See figure: 3.2

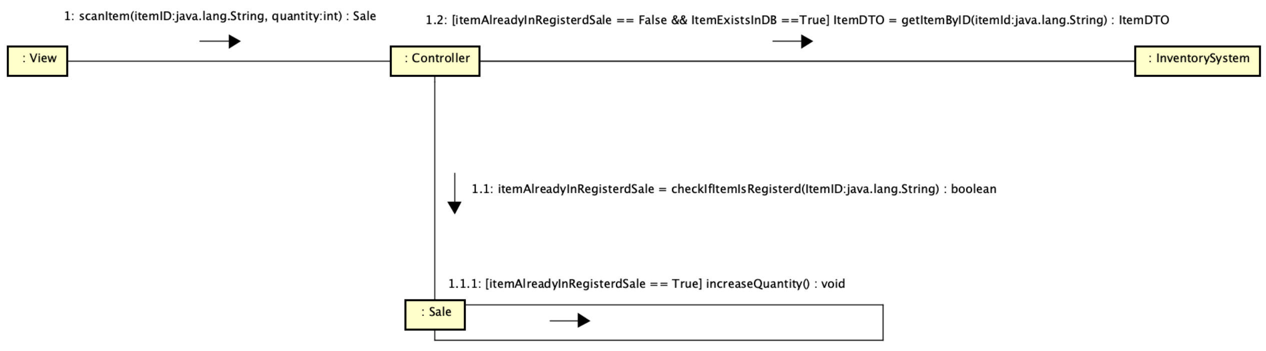
1: scanItem(itemID:java.lang.String, quantity:int) : Sale
1.2: [itemAlreadyInRegisterdSale == False && ItemExistsInDB ==True] ItemDTO = getItemByID(itemId:java.lang.String) : ItemDTO
: View
: Controller
: InventorySystem
1.1: itemAlreadyInRegisterdSale = checkIfItemIsRegisterd(ItemID:java.lang.String) : boolean
1.1.1: [itemAlreadyInRegisterdSale == True] increaseQuantity() : void
: Sale

Figure 3.2: The communication diagram for scanningGoods

Discount

When all items have been scanned we want to apply the discount to the sale.The cashier enters the customers ID in the view and the view sends the operation discountControl with the argument customerID to the Controller. The controler then sends a operation to the intergation layer where the discount database is. The discount database has the operation getDiscount to retrive the discount depending on the customerID. This is then stored in the variable discountAmount that is sent to Sale to be applied to the total price. See figure: 3.3

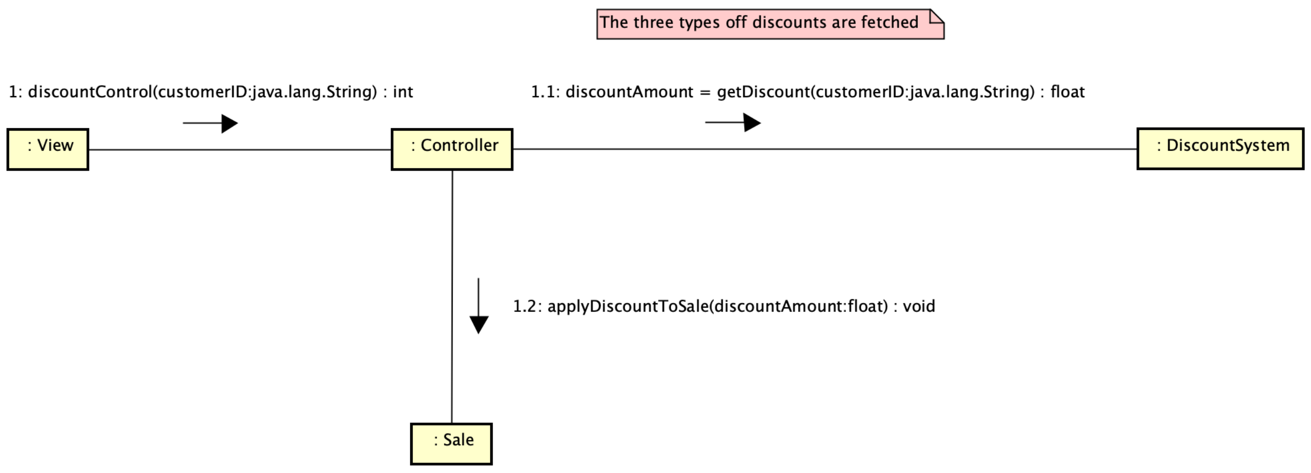
The three types off discounts are fetched
1: discountControl(customerID:java.lang.String) : int
1.1: discountAmount = getDiscount(customerID:java.lang.String) : float
: View
: Controller
: DiscountSystem
1.2: applyDiscountToSale(discountAmount:float) : void
: Sale

Figure 3.3: The communication diagram for discount

Payment

When the discount has been added we come to the payment. The cashier gets the payment from the customer and enters the amount in the view. The View calls the Controller with the operator payForGoods and the argument for this operation is the payment from the customer. The operation updates payment then acts on the sale and the sale calls on customerPayment. We then take the payment from the customer remove the total price and then we get the return witch is the change for the customer. See figure: 3.4

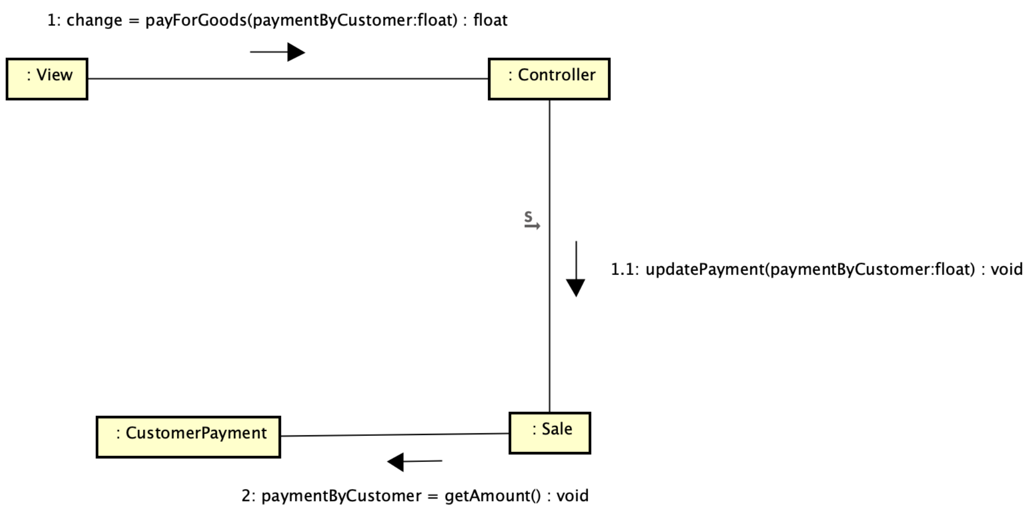
1: change = payForGoods(paymentByCustomer:float) : float
: View
: Controller
1.1: updatePayment(paymentByCustomer:float) : void
: CustomerPayment
: Sale
2: paymentByCustomer = getAmount() : void

Figure 3.4: The communication diagram for payment

End Sale

The cashier ends the sale. This is sent from View to the controller. We sent the sale (that contains the Items, total price, total discount, VAT and the time of the sale) to the accounting system. And then we send the sales to the inventory system to update the current inventory. The sales information is then made into a receipt that is printed for the customer. The operator updateInventory system is a public class within the Inventory system to make sure it can be called by the controller. See figure: 3.5

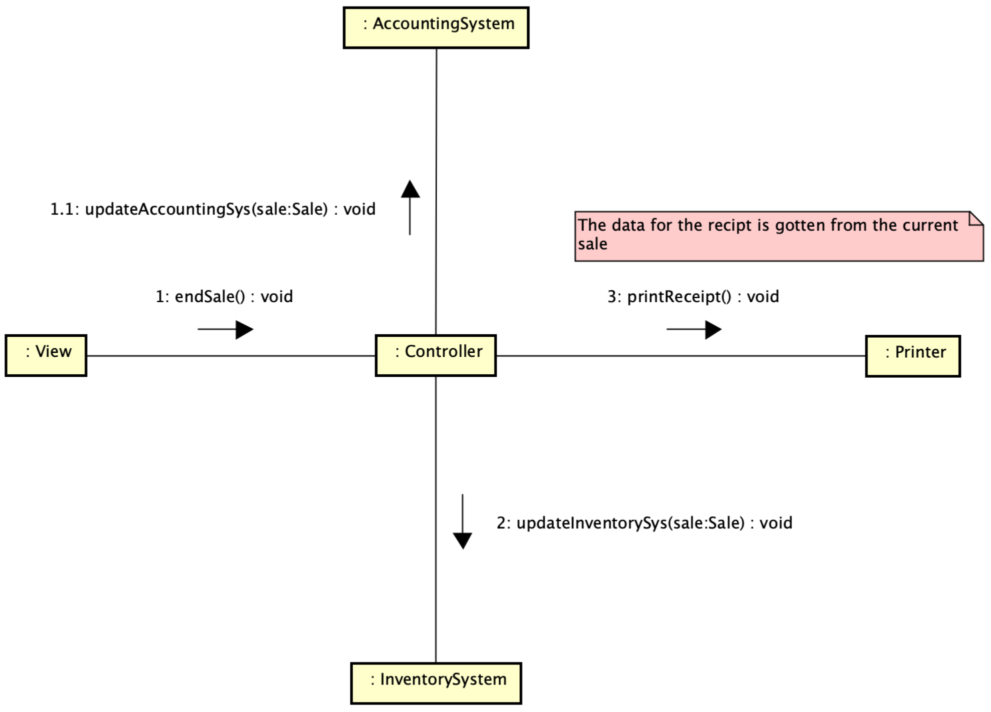
: AccountingSystem
1.1: updateAccountingSys(sale:Sale) : void
The data for the recipt is gotten from the current sale
1: endSale() : void
3: printReceipt() : void
: View
: Controller
: Printer
2: updateInventorySys(sale:Sale) : void
: InventorySystem

Figure 3.5: The communication diagram for endSale

Class Diagram

The startup is the main module that starts the view and the controller. The view then gives operations directly to the controller for example the startSale() and endSale(). The controller also has connections to all the classes in the integration since all the calls to the databases and the other external systems are made by the controller. The controller also has a connection to the class Sale. The Sale is connected to the item in the way that the sale contain the item and the ItmeDTO. The sale is also connected to the receipt since we get the information for the receipt from the Sale and to the Customer payment since we need to access a running total and during the payment we need to be able to receive a change. See figure: 3.6

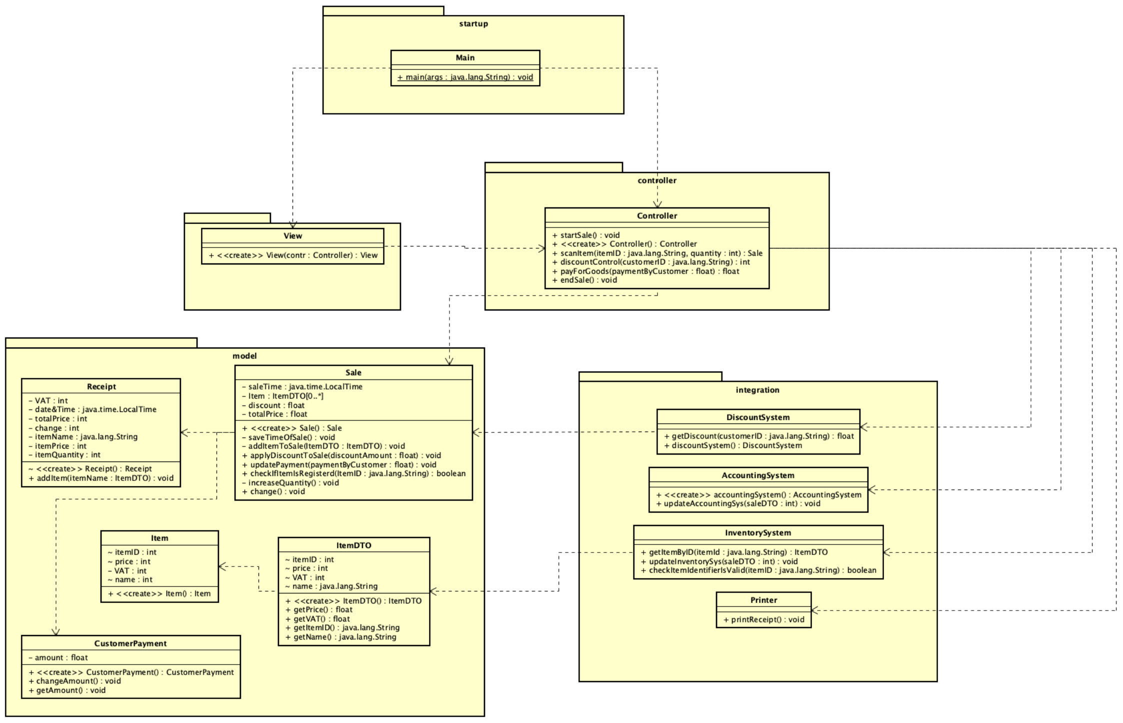
startup
Main
+ main(args : java.lang.String) : void
controller
Controller
+ startSale() : void
+ <<create>> Controller() : Controller
+ scanItem(itemID : java.lang.String, quantity : int) : Sale
+ discountControl(customerID : java.lang.String) : int
+ payForGoods(paymentByCustomer : float) : float
+ endSale() : void
View
+ <<create>> View(contr : Controller) : View
model
Receipt
- VAT : int
- date&Time : java.time.LocalTime
- totalPrice : int
- change : int
- itemName : java.lang.String
- itemPrice : int
- itemQuantity : int
~ <<create>> Receipt() : Receipt
+ addItem(itemName : ItemDTO) : void
Sale
- saleTime : java.time.LocalTime
- Item : ItemDTO[0..*]
- discount : float
- totalPrice : float
+ <<create>> Sale() : Sale
- saveTimeOfSale() : void
- addItemToSale(ItemDTO : ItemDTO) : void
+ applyDiscountToSale(discountAmount : float) : void
+ updatePayment(paymentByCustomer : float) : void
+ checkIfItemIsRegisterd(ItemID : java.lang.String) : boolean
- increaseQuantity() : void
+ change() : void
Item
~ itemID : int
~ price : int
- VAT : int
~ name : int
+ <<create>> Item() : Item
ItemDTO
~ itemID : int
~ price : int
~ VAT : int
~ name : java.lang.String
+ <<create>> ItemDTO() : ItemDTO
+ getPrice() : float
+ getVAT() : float
+ getItemID() : java.lang.String
+ getName() : java.lang.String
CustomerPayment
- amount : float
+ <<create>> CustomerPayment() : CustomerPayment
+ changeAmount() : void
+ getAmount() : void
integration
DiscountSystem
+ getDiscount(customerID : java.lang.String) : float
+ discountSystem() : DiscountSystem
AccountingSystem
+ <<create>> accountingSystem() : AccountingSystem
+ updateAccountingSys(saleDTO : int) : void
InventorySystem
+ getItemByID(itemId : java.lang.String) : ItemDTO
+ updateInventorySys(saleDTO : int) : void
+ checkItemIdentifierIsValid(itemID : java.lang.String) : boolean
Printer
+ printReceipt() : void

Figure 3.6: The final class diagram

4 Discussion

Understanding I do feel like the diagrams are well-structured Clear labeling, consistent notation, and meaningful class names contribute to better understanding.
MVC and Layer Patterns The diagram follows the MVC pattern. Layers (such as Model, View, and Controller) are shown in the model. The View handles user input in this case from the cashier and the Controller makes important operation calls to other classes while the model encapsulates business logic.
Coupling, Cohesion, and Encapsulation Low Coupling: The diagram doesn't explicitly show dependencies between classes. High Cohesion: Classes are focused on specific responsibilities (e.g., Receipt, Item) Encapsulation: Attributes and methods are encapsulated within classes.
Parameters, Return Values, and Types Let's take the example of the communication diagram Scanning goods. here i am using a return value that is a boolean to determine if an item has already been scanned this variable is then used in the guard to check if we should get information from the inventory system where we also get a return value that is stored in a DTO. A good example of were types have been used in a smart way is on the discount diagram. Here i have used a float since discounts are usually not whole numbers.
Objects vs. Primitive Data The diagram primarily shows classes (objects) rather than primitive data types. We have the DTO for Item to not have to have alot of duplicate data indifferent layers.
Java Naming Conventions Java conventions are followed for the most part. Im sure i missed it in some places.