Seminar 2

Object-Oriented Design, IV1350

Ayda Ghalkhanbaz
aydag@kth.se

April 17, 2024

Contents

1 Introduction

The second task of the retail store project includes designing a class diagram along with its communication diagrams indicating the system operations in detail. It is a buildup on the first task and shows the actions during a sale process.

This report covers the design of classes, methods and intercommunications between the two of them. The design was modelled using UML diagrams, namely class and communication diagrams. The results then will be shown and discussed in the later sections. I cooperated with Naveed Rahman and Nora Say to accomplish this task.

2 Method

The methods used for both the class and communication diagrams were explained thoroughly in chapter 5 of the book “Object Oriented Development”, but a short explanation of the used methods are as follows.

The first step was to use MVC and layer patterns to proceed from SSD and domain model (from previous seminar) to class and communication diagrams. The three base subsystems, View, Controller and Model were created at first then during the process more layers such as integration and startup were added to the class diagram.

The second step involved creating communication systems for each system operation based on the actions illustrated in the SSD. In other words, for each action taken in the SSD between the cashier and the system, a diagram was created to show the interaction in detail, including the inputs from the user interface and outputs from the system.

As the number of layers and classes increased, the more data was transferred between them, which caused a long parameter list when calling a new method. To avoid this, a couple of DTO’s (Data Transfer Object) were created to ease the data transfer between layers and classes. Every communication diagram was double checked at the end of design of each so that they have high cohesion (that all components should work in a way to form a coherent whole), low coupling (minimising dependencies between classes) and are encapsulated (irrelevant details are hidden to the user interface by making them private).

As the last step of the process, a class diagram was drawn and the connections between classes were drawn by arrows.

3 Result

Figure 3.1 to 3.6 illustrates the design of all communication diagrams while figure 3.7 shows how all classes are connected to each other in a class diagram.

The communication diagrams show each system operation in detail, where all connections are illustrated with an arrow, the method's name, its parameters and the return type of it.

For instance figure 3.2, the process of scanning and registering items is shown. The process starts by View calling the `registerItem()` method in the controller (i.e. when the cashier scans an item). The controller then will check the current sale to see whether the item has already been scanned. If so, the controller already knows all information about the item and doesn't need to fetch the item information from the inventory system. So it simply calls the method `updateItemInSale()` from the Sale class to increase the quantity of the scanned item.

However, if the item is not found in the sale, the controller needs to get the item information before updating the sale. So it calls the method `getItemInfo()` from the inventory system. After fetching the data, the controller can now add the item to the sale. Sale will then update the customer's shopping bag.

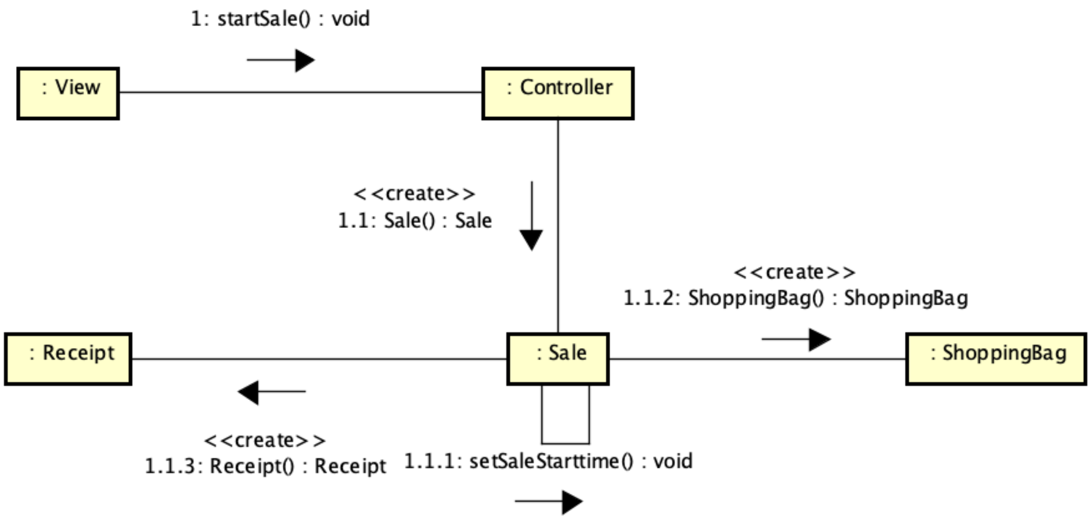

Figure 3.1: Communication diagram illustrating start sale process

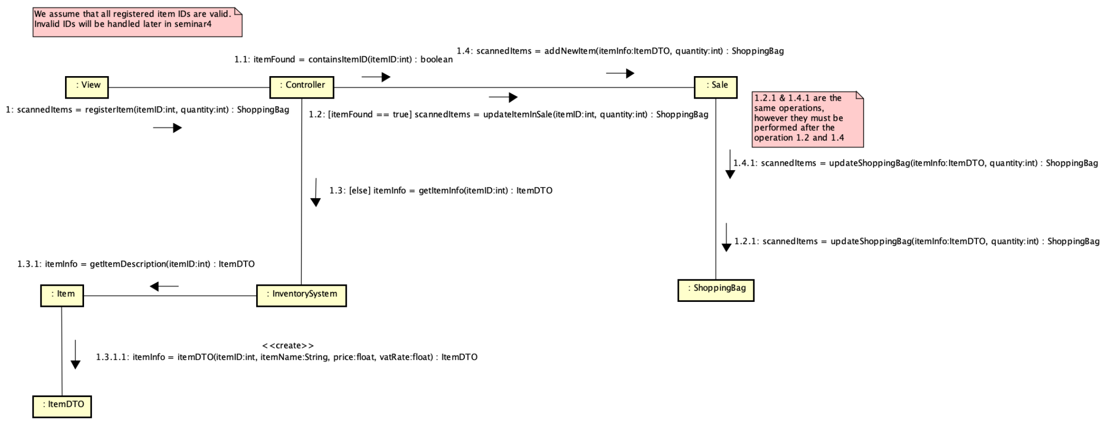
We assume that all registered item IDs are valid. Invalid IDs will be handled later in seminar4
1.1: itemFound = containsItemID(itemID:int) : boolean
1.4: scannedItems = addNewItem(itemInfo:ItemDTO, quantity:int) : ShoppingBag
: View
: Controller
: Sale
1: scannedItems = registerItem(itemID:int, quantity:int) : ShoppingBag
1.2: [itemFound == true] scannedItems = updateItemInSale(itemID:int, quantity:int) : ShoppingBag
1.2.1 & 1.4.1 are the same operations, however they must be performed after the operation 1.2 and 1.4
1.4.1: scannedItems = updateShoppingBag(itemInfo:ItemDTO, quantity:int) : ShoppingBag
1.3: [else] itemInfo = getItemInfo(itemID:int) : ItemDTO
1.2.1: scannedItems = updateShoppingBag(itemInfo:ItemDTO, quantity:int) : ShoppingBag
1.3.1: itemInfo = getItemDescription(itemID:int) : ItemDTO
: Item
: InventorySystem
: ShoppingBag
<<create>>
1.3.1.1: itemInfo = itemDTO(itemID:int, itemName:String, price:float, vatRate:float) : ItemDTO
: ItemDTO

Figure 3.2: Communication diagram illustrating register item process

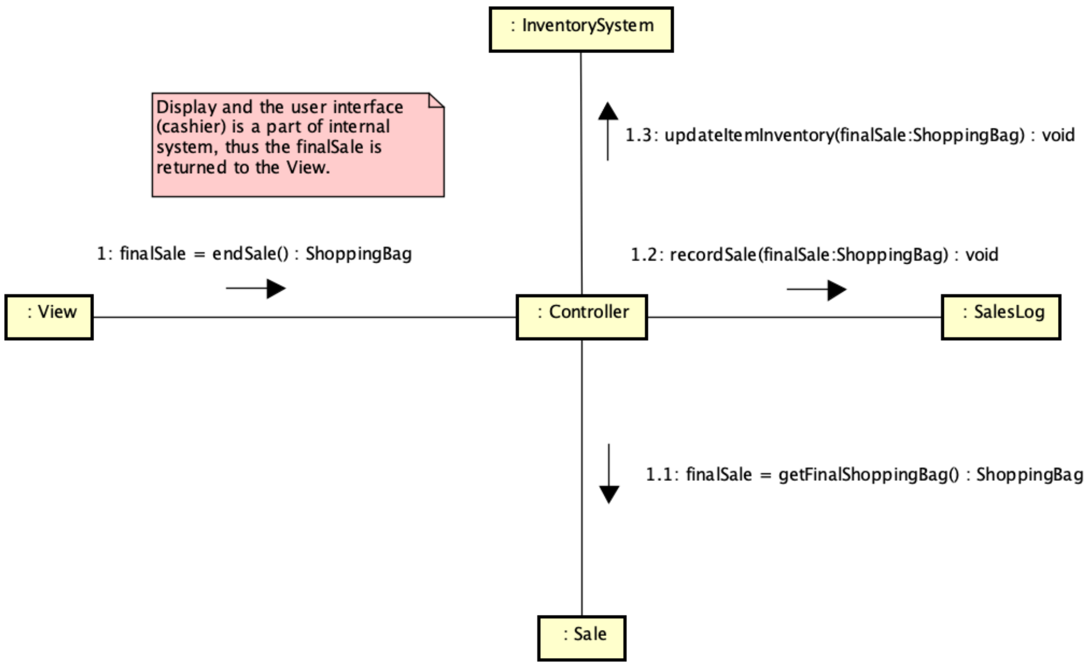
: InventorySystem
Display and the user interface (cashier) is a part of internal system, thus the finalSale is returned to the View.
1.3: updateItemInventory(finalSale:ShoppingBag) : void
1: finalSale = endSale() : ShoppingBag
1.2: recordSale(finalSale:ShoppingBag) : void
: View
: Controller
: SalesLog
1.1: finalSale = getFinalShoppingBag() : ShoppingBag
: Sale

Figure 3.3: Communication diagram illustrating end sale process

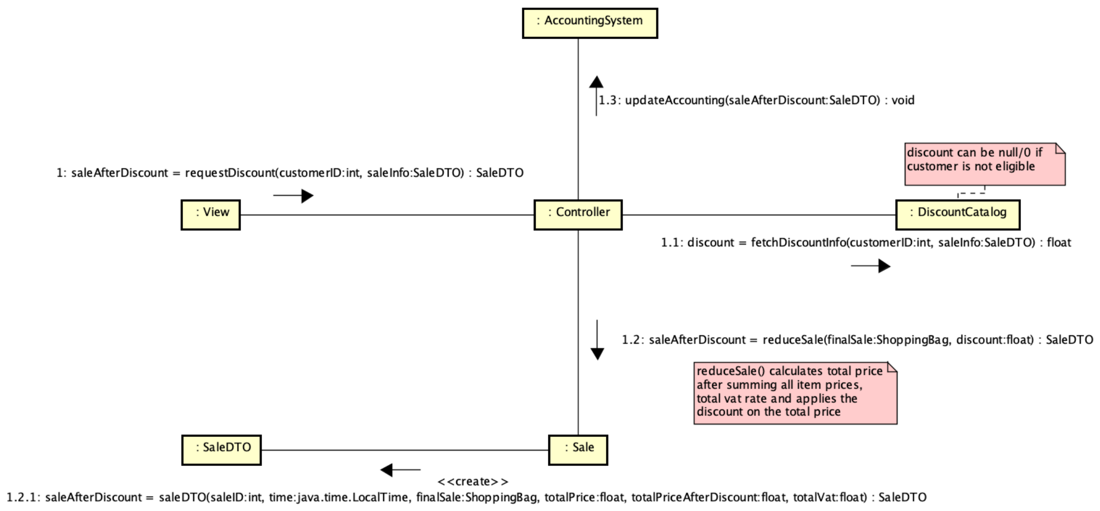
: AccountingSystem
1.3: updateAccounting(saleAfterDiscount:SaleDTO) : void
discount can be null/0 if customer is not eligible
1: saleAfterDiscount = requestDiscount(customerID:int, saleInfo:SaleDTO) : SaleDTO
: View
: Controller
: DiscountCatalog
1.1: discount = fetchDiscountInfo(customerID:int, saleInfo:SaleDTO) : float
1.2: saleAfterDiscount = reduceSale(finalSale:ShoppingBag, discount:float) : SaleDTO
reduceSale() calculates total price after summing all item prices, total vat rate and applies the discount on the total price
: SaleDTO
: Sale
<<create>>
1.2.1: saleAfterDiscount = saleDTO(saleID:int, time:java.time.LocalTime, finalSale:ShoppingBag, totalPrice:float, totalPriceAfterDiscount:float, totalVat:float) : SaleDTO

Figure 3.4: Communication diagram illustrating discount process

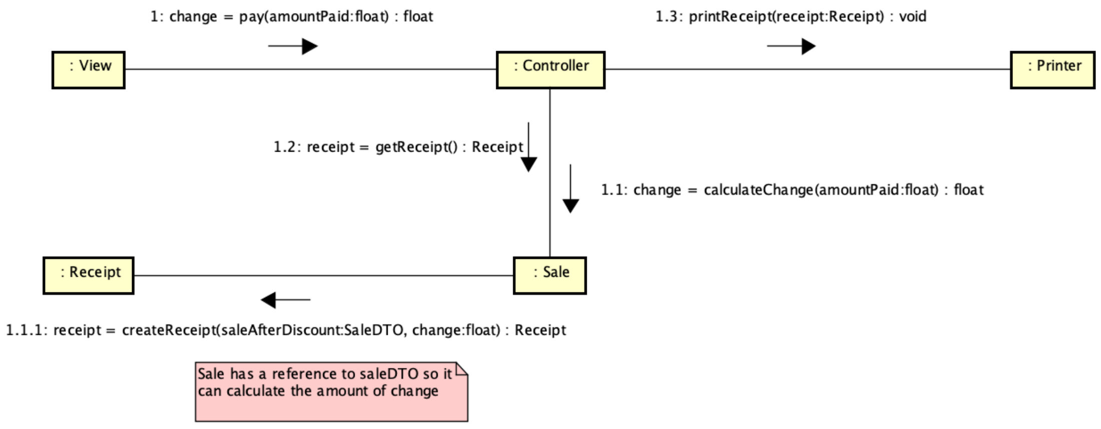
1: change = pay(amountPaid:float) : float
1.3: printReceipt(receipt:Receipt) : void
: View
: Controller
: Printer
1.2: receipt = getReceipt() : Receipt
1.1: change = calculateChange(amountPaid:float) : float
: Receipt
: Sale
1.1.1: receipt = createReceipt(saleAfterDiscount:SaleDTO, change:float) : Receipt
Sale has a reference to saleDTO so it can calculate the amount of change

Figure 3.5: Communication diagram illustrating payment process

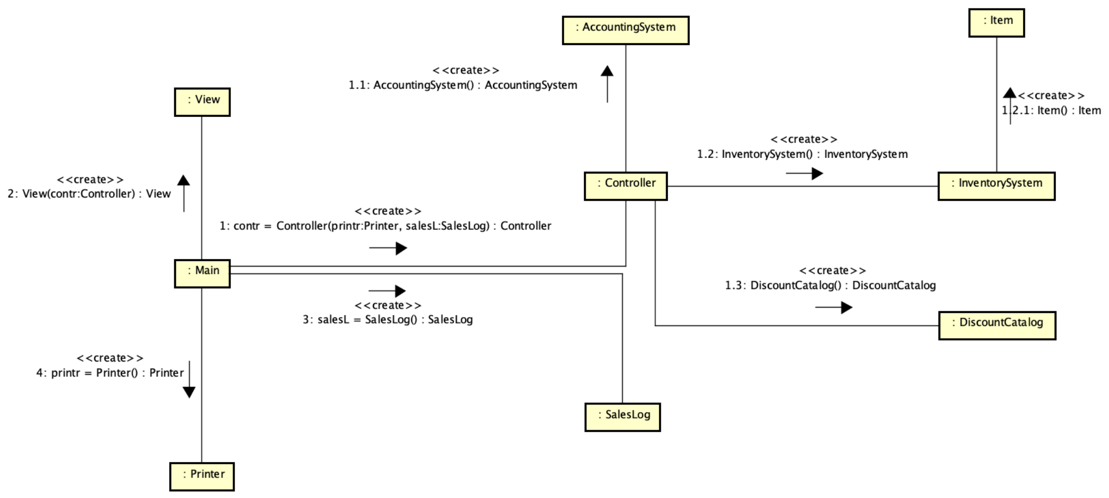
: AccountingSystem
: Item
<<create>>
1.1: AccountingSystem() : AccountingSystem
<<create>>
1.2.1: Item() : Item
: View
<<create>>
1.2: InventorySystem() : InventorySystem
<<create>>
2: View(contr:Controller) : View
: Controller
: InventorySystem
<<create>>
1: contr = Controller(printr:Printer, salesL:SalesLog) : Controller
: Main
<<create>>
1.3: DiscountCatalog() : DiscountCatalog
<<create>>
3: salesL = SalesLog() : SalesLog
: DiscountCatalog
<<create>>
4: printr = Printer() : Printer
: SalesLog
: Printer

Figure 3.6: Communication diagram illustrating startup process

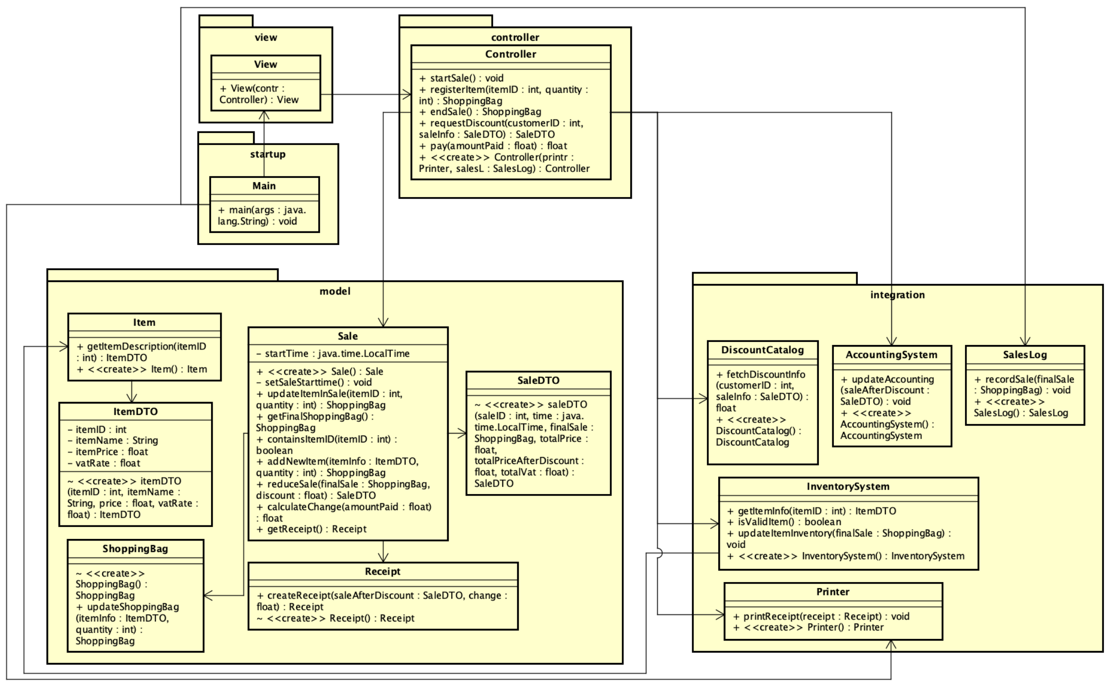
view
View
+ View(contr : Controller) : View
controller
Controller
+ startSale() : void
+ registerItem(itemID : int, quantity : int) : ShoppingBag
+ endSale() : ShoppingBag
+ requestDiscount(customerID : int, saleInfo : SaleDTO) : SaleDTO
+ pay(amountPaid : float) : float
+ <<create>> Controller(printr : Printer, salesL : SalesLog) : Controller
startup
Main
+ main(args : java.lang.String) : void
model
Item
+ getItemDescription(itemID : int) : ItemDTO
+ <<create>> Item() : Item
ItemDTO
- itemID : int
- itemName : String
- itemPrice : float
- vatRate : float
~ <<create>> itemDTO (itemID : int, itemName : String, price : float, vatRate : float) : ItemDTO
ShoppingBag
~ <<create>> ShoppingBag() : ShoppingBag
+ updateShoppingBag (itemInfo : ItemDTO, quantity : int) : ShoppingBag
Sale
- startTime : java.time.LocalTime
+ <<create>> Sale() : Sale
- setSaleStarttime() : void
+ updateItemInSale(itemID : int, quantity : int) : ShoppingBag
+ getFinalShoppingBag() : ShoppingBag
+ containsItemID(itemID : int) : boolean
+ addNewItem(itemInfo : ItemDTO, quantity : int) : ShoppingBag
+ reduceSale(finalSale : ShoppingBag, discount : float) : SaleDTO
+ calculateChange(amountPaid : float) : float
+ getReceipt() : Receipt
SaleDTO
~ <<create>> saleDTO (saleID : int, time : java.time.LocalTime, finalSale : ShoppingBag, totalPrice : float, totalPriceAfterDiscount : float, totalVat : float) : SaleDTO
Receipt
+ createReceipt(saleAfterDiscount : SaleDTO, change : float) : Receipt
~ <<create>> Receipt() : Receipt
integration
DiscountCatalog
+ fetchDiscountInfo (customerID : int, saleInfo : SaleDTO) : float
+ <<create>> DiscountCatalog() : DiscountCatalog
AccountingSystem
+ updateAccounting (saleAfterDiscount : SaleDTO) : void
+ <<create>> AccountingSystem() : AccountingSystem
SalesLog
+ recordSale(finalSale : ShoppingBag) : void
+ <<create>> SalesLog() : SalesLog
InventorySystem
+ getItemInfo(itemID : int) : ItemDTO
+ isValidItem() : boolean
+ updateItemInventory(finalSale : ShoppingBag) : void
+ <<create>> InventorySystem() : InventorySystem
Printer
+ printReceipt(receipt : Receipt) : void
+ <<create>> Printer() : Printer

Figure 3.7: Class diagram illustrating the connections between classes

4 Discussion

The designed model for this task follows the MVC model pattern, which handles user interface, the logic and the data separately. The View class represents the user interface which according to the business rules of the project should not be coded as an external system and the provided model doesn't include any code for this layer.

The controller layer is responsible for creating a connection between the View and the Model layers. It has one important task, that is, calling the relevant methods from the model layer in correct order.

While the model layer takes care of the underlying logic behind the entire system. Furthermore the layers can access only the layer below themselves, for instance the view layer can only have contact with the controller, while the controller can make a connection to the model or the integration layer. However an integration layer cannot go upwards and make contact with the view layer itself.

The model also follows the three fundamental principles, i.e. low coupling, high cohesion and good encapsulation. For instance, there are no direct links between view and model and the only way for the View to connect to the Model is through the controller. This is evident by looking at both class diagram and communication diagrams.

The reason why I believe that the model has high cohesion is that classes functionality are pretty clear. As an example, the Sale() class has the responsibility to manage the sale information, updating saleslog and shopping bags, creating receipts etc.

As mentioned earlier, the model also has a good encapsulation. An example of this is shown in figure 3.7. As we can see, there are classes such as Receipt, SaleDTO, ItemDTO and ShoppingBag that are only visible inside that specific layer and cannot be accessed by an outer layer like controller.

Also, the model covers multiple object types, specifically DTO objects, to minimise the amount of parameter lists, sending forth and back. There is neither any static method or class.

As we can see in the diagrams, all Java naming conventions are followed. For example, all class names (such as: Sale, DiscountCatalog, AccountingSystem, etc.) followers an upper CamelCase, While the methods are written in lower camelCase (such as: fetchDiscountInfo, calculateChange, etc.).