Seminar 2

Object-Oriented Design, IV1350

Jonathan Nilsson Cullgert jonnc@kth.se

7 April 2025

Project Members:
[Jonathan Nilsson Cullgert, jonnc@kth.se]
[Joel Friis, joefri@kth.se]
[Malte Berg, malte@malte.nu]

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

In this assignment a object-oriented program is designed by applying object-oriented architecture that follows the flow of a sale from the previous assignment. For this seminar no code was written, only the design of the program was created. To achieve a good object-oriented design and architecture the Model-View-Controller (MVC) principle was followed as well as the layer pattern. To handles objects of data, data transfer objects (DTO) can be created to flow between classes.

The design of the program was designed as mentioned previously from the task in seminar 1. The SSD from that task is translated into a series of communication diagrams and a class diagram. If it is necessary some changes might be made from the SSD to make the diagrams in this assignment correct. The important part in this assignment is to follow MVC and the layer principle as well as having good encapsulation, high cohesion and low coupling.

2 Method

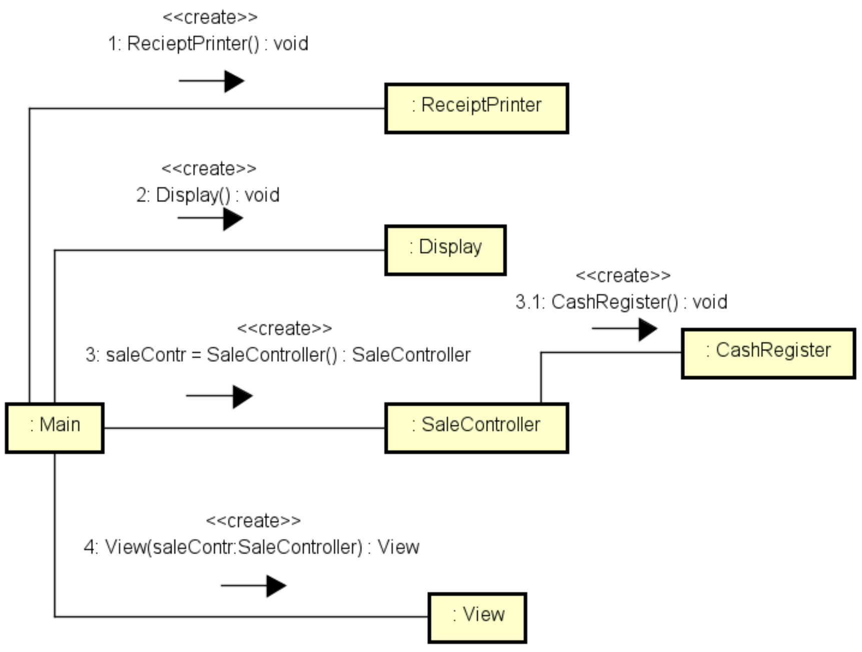
This assignment was also solved in Astah. The first step in creating or diagrams was by following our SSD and the basic/alternate flow and making small changes to our SSD to better fit the instructed flow. The first step was creating the startup, main class which creates all our internal and external systems the view, controller, register, display and receipt printer.

After the main class has started everything we assume that a sale is being started by the cashier and customer in which the sale controller starts a new sale of the sale class. From there a new diagram was created to show how items are scanned. The view calls scanProduct from the controller which retrieves the price from the external inventory and updates the price internally as well as on the display.

After all products have been scanned the view can call the endSale from the controller which ends the sale and retrieves the running total plus vat from the sale. When the sale has ended the customer can flag for a discount which is a new diagram. The view calls getdiscount with a customer id. The controller will retrieve the current sale as a DTO and transfer that with the customer id to the external discount database. That way the database can handle calculations of the discount and return the amount discounted from the sale. The controller will wait for the amount discounted and call updatediscount from the sale with the amount discounted.

When the discount has been added the payment will take place which is the final diagram. The customer will pay with the enteramount method and eventually receive a receipt and change. The controller will receive the final sale as DTO from sale which it will send to the external inventory and accounting system to update them. It will also be sent with the paid amount to the register to update cash in register as well as return change. The change and paid amount will be sent to sale which in turn will create a receipt object with the DTO, paid amount, change and current time/date. This receipt will be sent to the receipt printer which will print the receipt.

3 Result



<<create>>
1: RecieptPrinter() : void
: ReceiptPrinter
<<create>>
2: Display() : void
: Display
<<create>>
3.1: CashRegister() : void
: CashRegister
<<create>>
3: saleContr = SaleController() : SaleController
: Main
: SaleController
<<create>>
4: View(saleContr:SaleController) : View
: View

Figure 1: The communication diagram which shows how the startup of the program is done.

In 1 Main creates all internal and some external systems. The controller will create the register which is connected to this specific sale controller. The view will also be created with the current sale controller as an input argument as that controller will control the view.

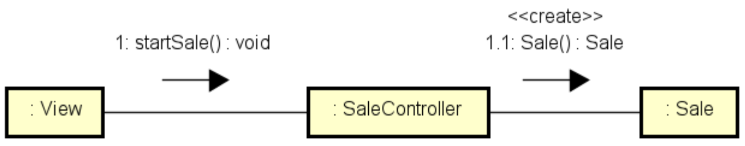


Figure 2: The communication diagram which shows how a sale is started by the customer & cashier.

In 2 the view will call the startSale() method in the controller when a sale is started. The controller will then start a new sale from the sale class. This will be a new instance of a sale.

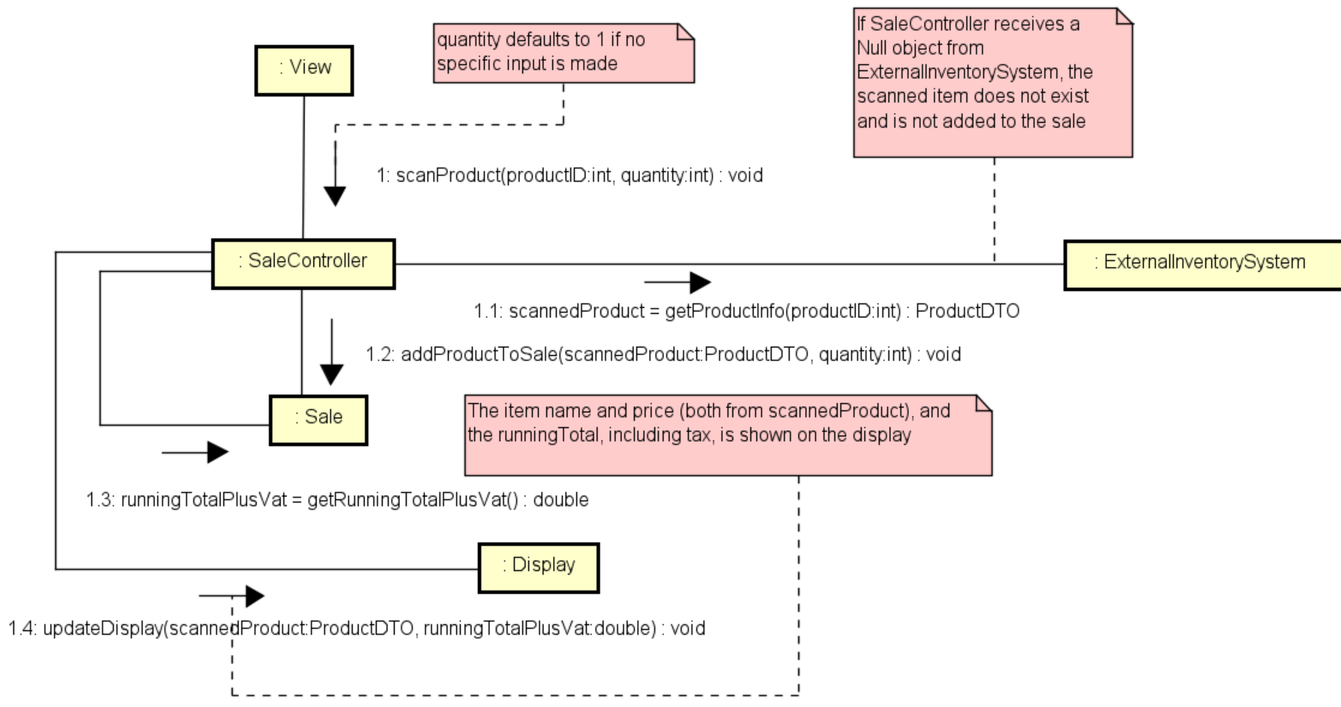

: View
quantity defaults to 1 if no specific input is made
If SaleController receives a Null object from ExternalInventorySystem, the scanned item does not exist and is not added to the sale
1: scanProduct(productID:int, quantity:int) : void
: SaleController
: ExternalInventorySystem
1.1: scannedProduct = getProductInfo(productID:int) : ProductDTO
1.2: addProductToSale(scannedProduct:ProductDTO, quantity:int) : void
: Sale
The item name and price (both from scannedProduct), and the runningTotal, including tax, is shown on the display
1.3: runningTotalPlusVat = getRunningTotalPlusVat() : double
: Display
1.4: updateDisplay(scannedProduct:ProductDTO, runningTotalPlusVat:double) : void

Figure 3: The communication diagram as a sale is taking place and all the items are being scanned, information is retrieved and displays are updated.

The diagram in 3 is a bit more complex than the previous two diagrams. The view will call scanProduct() in the controller with the id and quantity as parameters. The controller will then retrieve the item as a DTO from the inventory system (if it is valid) and will then add the product with the correct amount in to the sale. The current total price with VAT will be calculated and sent to the display with updateDisplay.

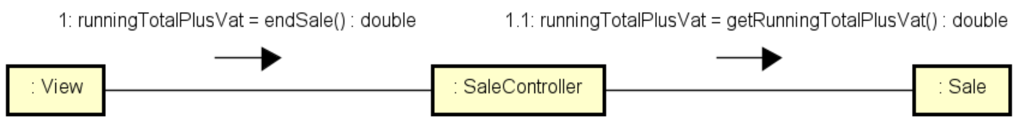

1: runningTotalPlusVat = endSale() : double
1.1: runningTotalPlusVat = getRunningTotalPlusVat() : double
: View
: SaleController
: Sale

Figure 4: The communication diagram which shows how the end of the program is done.

Eventually the sale will end when there are no more products to scan. The cashier or customer will endSale from the view which will through the controller get the running total plus VAT as depicted in 4.

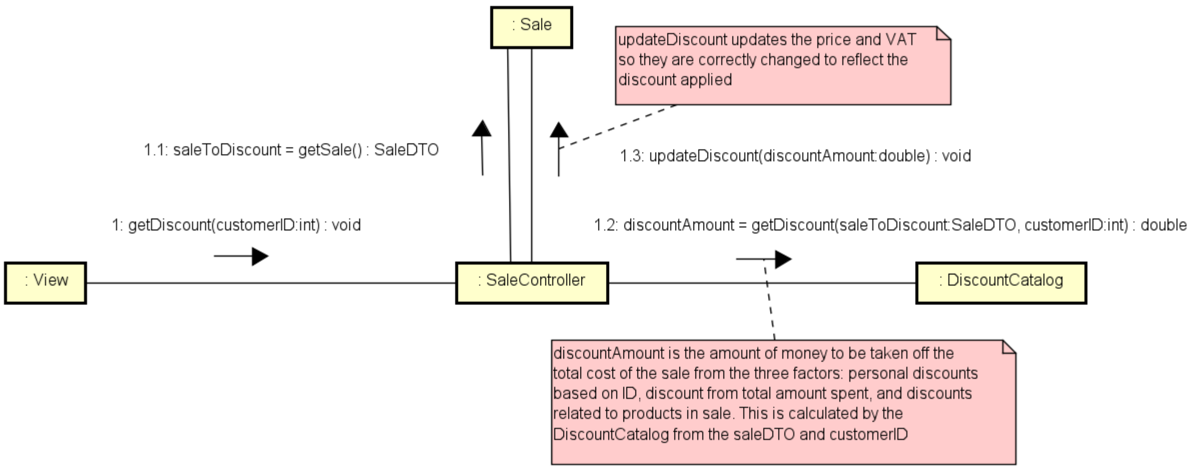

: Sale
updateDiscount updates the price and VAT so they are correctly changed to reflect the discount applied
1.1: saleToDiscount = getSale() : SaleDTO
1.3: updateDiscount(discountAmount:double) : void
1: getDiscount(customerID:int) : void
1.2: discountAmount = getDiscount(saleToDiscount:SaleDTO, customerID:int) : double
: View
: SaleController
: DiscountCatalog
discountAmount is the amount of money to be taken off the total cost of the sale from the three factors: personal discounts based on ID, discount from total amount spent, and discounts related to products in sale. This is calculated by the DiscountCatalog from the saleDTO and customerID

Figure 5: The communication diagram that pictures the necessary method calls if a discount is attempted to be applied to the sale.

In 5 the customer flags that they want a discount with their personal id. The controller will retrieve the current sale as a DTO and send that to the external discount database with the id for calculations and validity of a discount for the customer. If the customer is eligible for a discount it is returned as the amount reduced of the current total. The controller will then update the total of the current sale.

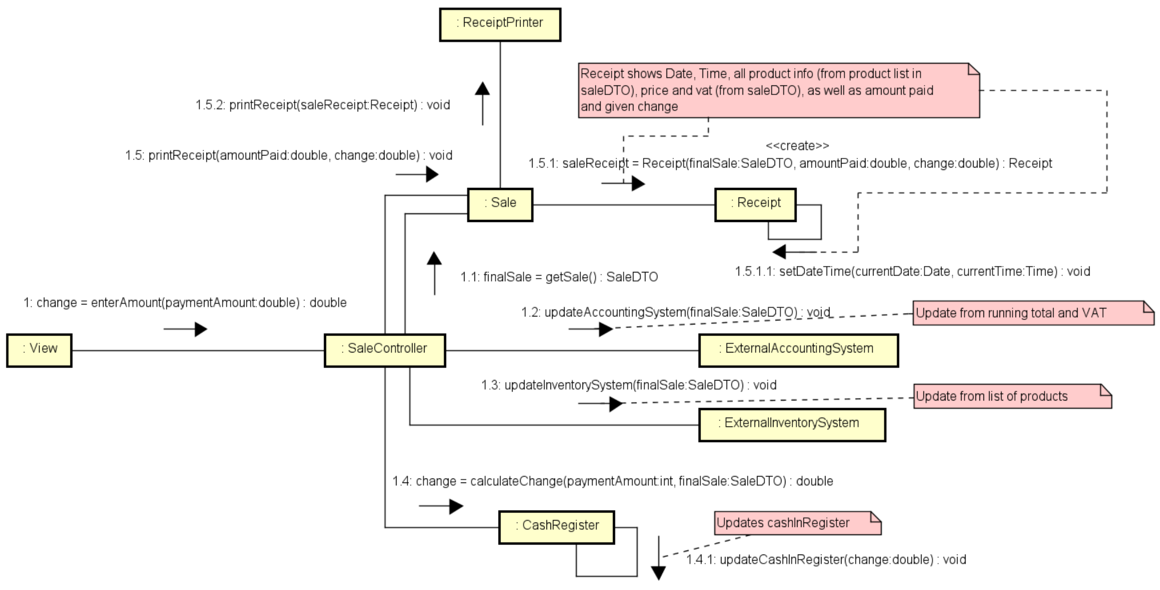

: ReceiptPrinter
Receipt shows Date, Time, all product info (from product list in saleDTO), price and vat (from saleDTO), as well as amount paid and given change
1.5.2: printReceipt(saleReceipt:Receipt) : void
1.5: printReceipt(amountPaid:double, change:double) : void
<<create>>
1.5.1: saleReceipt = Receipt(finalSale:SaleDTO, amountPaid:double, change:double) : Receipt
: Sale
: Receipt
1.5.1.1: setDateTime(currentDate:Date, currentTime:Time) : void
1.1: finalSale = getSale() : SaleDTO
1: change = enterAmount(paymentAmount:double) : double
1.2: updateAccountingSystem(finalSale:SaleDTO) : void
Update from running total and VAT
: View
: SaleController
: ExternalAccountingSystem
1.3: updateInventorySystem(finalSale:SaleDTO) : void
Update from list of products
: ExternalInventorySystem
1.4: change = calculateChange(paymentAmount:int, finalSale:SaleDTO) : double
: CashRegister
Updates cashInRegister
1.4.1: updateCashInRegister(change:double) : void

Figure 6: The communication diagram that shows how a payment is done and how all the external systems are updated with the correct information to close the sale.

In 6 the controller receives the amount paid by the customer. The controller will then create a DTO of the final sale, send that information to inventory and accounting to update them. It will calculate the change and update cash in register with the amount paid and the saleDTO. The sale will print the receipt by creating a new receipt object with amount paid, change and current time and send that object to the receipt printer.

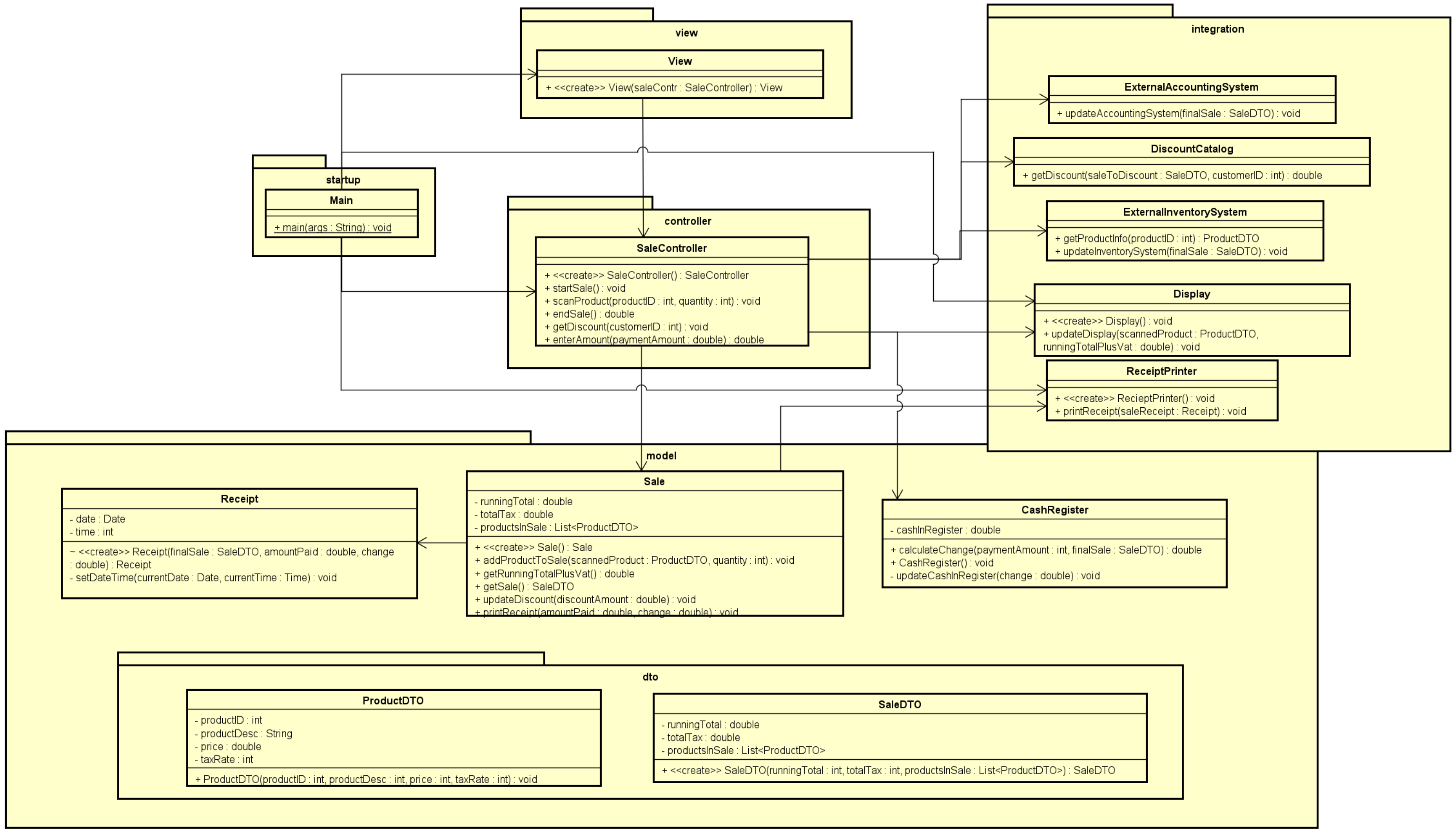

view
View
+ <<create>> View(saleContr : SaleController) : View
startup
Main
+ main(args : String) : void
controller
SaleController
+ <<create>> SaleController() : SaleController
+ startSale() : void
+ scanProduct(productID : int, quantity : int) : void
+ endSale() : double
+ getDiscount(customerID : int) : void
+ enterAmount(paymentAmount : double) : double
integration
ExternalAccountingSystem
+ updateAccountingSystem(finalSale : SaleDTO) : void
DiscountCatalog
+ getDiscount(saleToDiscount : SaleDTO, customerID : int) : double
ExternalInventorySystem
+ getProductInfo(productID : int) : ProductDTO
+ updateInventorySystem(finalSale : SaleDTO) : void
Display
+ <<create>> Display() : void
+ updateDisplay(scannedProduct : ProductDTO, runningTotalPlusVat : double) : void
ReceiptPrinter
+ <<create>> RecieptPrinter() : void
+ printReceipt(saleReceipt : Receipt) : void
model
Receipt
- date : Date
- time : int
~ <<create>> Receipt(finalSale : SaleDTO, amountPaid : double, change : double) : Receipt
- setDateTime(currentDate : Date, currentTime : Time) : void
Sale
- runningTotal : double
- totalTax : double
- productsInSale : List<ProductDTO>
+ <<create>> Sale() : Sale
+ addProductToSale(scannedProduct : ProductDTO, quantity : int) : void
+ getRunningTotalPlusVat() : double
+ getSale() : SaleDTO
+ updateDiscount(discountAmount : double) : void
+ printReceipt(amountPaid : double, change : double) : void
CashRegister
- cashInRegister : double
+ calculateChange(paymentAmount : int, finalSale : SaleDTO) : double
+ CashRegister() : void
- updateCashInRegister(change : double) : void
dto
ProductDTO
- productID : int
- productDesc : String
- price : double
- taxRate : int
+ ProductDTO(productID : int, productDesc : int, price : int, taxRate : int) : void
SaleDTO
- runningTotal : double
- totalTax : double
- productsInSale : List<ProductDTO>
+ <<create>> SaleDTO(runningTotal : int, totalTax : int, productsInSale : List<ProductDTO>) : SaleDTO

Figure 7: The class diagram following both MVC and layer principle, showing how all classes are dependent on eachother.

The class diagram shows how all the classes are connected to each other and their dependencies. The integration systems are mostly dependant on the controller and the main class. The view and controller are started by the main class. The controller is dependant on cashregister and sale in the model. The two DTOs are used in several different layers in the model.

4 Discussion

I would say that the design in general is simplistic and easy to follow but the payment interaction diagram as well as the class diagram are a bit cluttered with much but necessary information. In the class diagram there is the MVC layers, as well as the startup layer and the integration layer. This follows both the MVC principle and the layer principle. The layers and packages are enough to implement the design without it being to cluttered and classes being in packages or layers that they should not be in. One could make the point that the DTOs should have their own package but we do not find it necessary to that since it is implied that multiple layers have access to the DTOs.

Again in general this solution has low coupling with the expception of maybe the saleDTO which is transferred to the external systems. But since a DTO is made to be transferred to other layers changes in it should not break other classes unless necessary information is removed from the DTO but in that case the fundamental task of the program is changed. The sale class could be argued has high coupling to the controller but yet again, the controller must be able to update the model and therefore rely on the model being somewhat stable and not changed too much. The design has high cohesion and good encapsulation, classes only relate to one specific task, whether that is to create a receipt or handle a sale. We have private attributes, public, private and package private methods which creates good encapsulation. We do not have any static methods since we do not have any classes where we want to call methods without creating objects, all classes must be created to call their methods.

We try to send objects as much as possible but sometimes primitive data must be sent, such as ids or amounts. The DTOs can never change their values once they have been created but we sometimes hand out attributes (without changing them!) such as in sale when getrunningtotal gives out the price which will then be sent to the display. We have tried to explain our parameters, return values and types as much as possible with good naming but in the cases where even we found it a bit confusing we have tried to clear it up by using notes.

References

- The Task description in canvas
- Lindbäck, L. (2024) ‘Chapter 5 Design’, in A First Course in Object-Oriented Development A Hands-On Approach, pp. 45-81. Available at: `https://e.pcloud.`

`link/publink/show?code=XZa6eKZI6J9qagkziJ7TiyNrMf9qQhMDrC7` (Accessed: 7 April 2025).

- Lindbäck, L. (2025) ‘ Föreläsning 5, Designprinciper, del 1 ’, 7 April.
- Lindbäck, L. (2025) ‘ Föreläsning 5, Designprinciper, del 2 ’, 7 April.
- Lindbäck, L. (2025) ‘ Föreläsning 6, arkitektur, del 1 ’, 7 April.
- Lindbäck, L. (2025) ‘ Föreläsning 6, arkitektur, del 2 ’, 7 April.
- Lindbäck, L. (2025) ‘ Föreläsning 7, designövning, första systemoperationen i uppgiften till sem2 ’, 7 April.