Seminar 2

Object-Oriented Design, IV1350

Adalet Adiljan, adalat@kth.se

16th of April 2023

Contents

1. Introduction

The purpose with the seminar 2 is to advance our previous model and diagram from the seminar 1 by creating a MVC and Layers diagram and a communication diagram (CCD) / system sequence diagram (SSD) to clarify the messages and requests between the entities. The choices between whether we wanted to use SSD or CCD for the interaction was freely up to our decision, although the purpose to show the content(s) of the communication would be easier to depict and understood in how the entities and classes are dependent on each other, with the consideration to achieving a high cohesion, low coupling and good encapsulation. The definition for these words are following: high cohesion means how much the objects are related to each other.

Low coupling means the level measure of dependency between the objects or classes, while the encapsulation stands for how protected and hidden the data is. Good encapsulation means that the data is protected in the interfaces, and is “exposed” in a welldefined interface.

MVC and Layers are two different types of architectural designs which include patterns to follow with the purpose to separate and organize the codes in a structural way. It is similarly used as opposed to how we would use an outline for the architectural design to build a house. They can both be applied depending on the requirements and characteristics of the application. MVC (Model-View-Controller) usually has three separate packages (subsystems) given by their name, to visually categorize and divide the codes depending on their purpose while Layers corresponds to the divided separate layers within its own responsibilities and dependencies (visualized as classes within the package). The report covers groups procedure of the assignment in the method chapter 2, while the result is presented and discussed in the result in chapter 3. The discussion regarding the given criteria in the assessment is held in chapter 4. The assignment has been contributed together in the group of 3 people in total, including *Ali Kazimov* & *Haron Osman.*

2. Method

The assignment has been solved by the help of numerous techniques that have mainly been covered in the recorded lectures for building the MVC and Layers design.

CCD diagram

When creating the corresponding CCD to our SSD, we can remember that it is an interaction diagram which serves the same purpose as an SSD, i.e. defining the messages between the objects. Despite the similarities, something that we had to consider is to build a separate CCD interaction for every system operation to avoid any potential incoherent or cryptic connections that could lower the chance of creating a high cohesion, low coupling and encapsulation of the objects when clarifying the connections later between the subsystems in the MVC and Layers model. It is suitable to choose SSD over CCD if you are dependent on the timing sequence of the events since it has a time axis going downwards whereas the CCD is more flexible with connecting one object with multiple objects in both the horizontal and vertical direction.

Another thing to consider when building the diagram is that it’s an iterative process where we need to expect creating new classes (objects) as result of curtain method operations which is also being added automatically to the curtain packages during the process. It is also iterative in the terms of potentially optimizing our existing SSD or DM if we notice that a curtain object has too many relations with multiple objects, which in turn creates the package it belongs to having relations or dependencies with multiple packages.

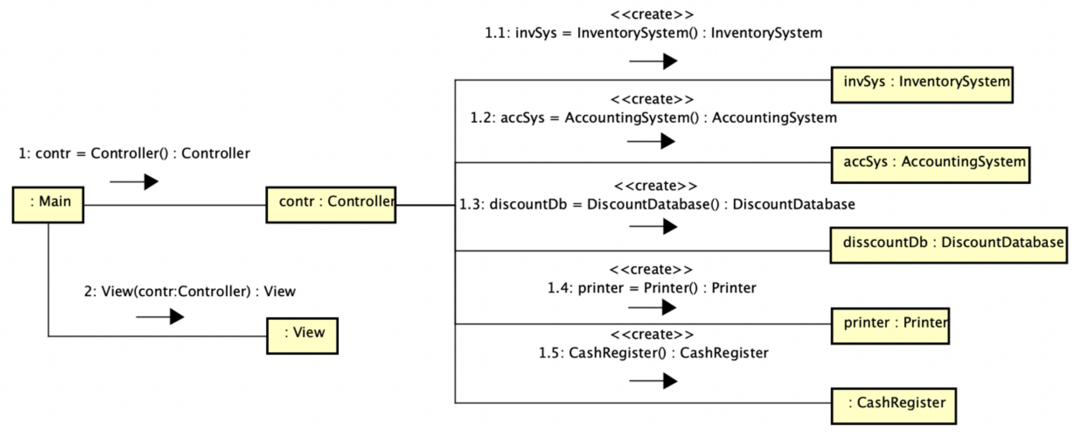
There are numerous of steps we considered when arriving to the solution of our CCDs. Firstly, we need to be analyzing the requirements that is needed for the system operation in question. Does the method return anything?

The last task after finishing defining the system operation mainly consisted of including the results of the interaction, by “moving” our diagram to our MVC and Layers model, where it then converted it to a code form. The last task also included to define the relation between different classes that happen to be in different packages, by drawing a clear arrow.

3. Result

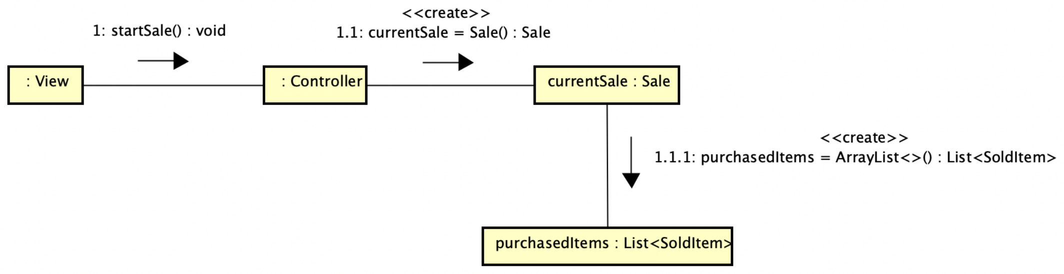
This chapter of the report presents the result of the given task. The task itself can be divided into multiple CCD diagrams, where each and one of the diagrams represents a system operation from the SSD. There are 5 operations in our sequence diagram, but depending on how advanced the system is, we can be needing multiple CCDs. In this case, we have implemented an own diagram for the if-statement that is included in our SSD. The end of the picture represents the result of our final MVC and Layers diagram after implementing / moving our CCDs into their own related package.

Image 1: When starting the sale, we are creating new instances of the classes in main, when the controller receives the request of starting the sale from main. The new instances of the external classes are created, along with the cash register. The arrows are indicating the direction of the message flow.



1: contr = Controller() : Controller
: Main
contr : Controller
2: View(contr:Controller) : View
: View
<<create>>
1.1: invSys = InventorySystem() : InventorySystem
invSys : InventorySystem
<<create>>
1.2: accSys = AccountingSystem() : AccountingSystem
accSys : AccountingSystem
<<create>>
1.3: discountDb = DiscountDatabase() : DiscountDatabase
disscountDb : DiscountDatabase
<<create>>
1.4: printer = Printer() : Printer
printer : Printer
<<create>>
1.5: CashRegister() : CashRegister
: CashRegister

Image2:
In this operation, we are creating a new instance of the sale to be able to save all of the relevant items and save them in the arraylist of all sold items. The arrow indicates the direction of the message flow.



1: startSale() : void
: View
: Controller
<<create>>
1.1: currentSale = Sale() : Sale
currentSale : Sale
<<create>>
1.1.1: purchasedItems = ArrayList<>() : List<SoldItem>
purchasedItems : List<SoldItem>

on:

Image 3:
In the third image, we are seeing the interaction between the classes when the goods has been purchased.

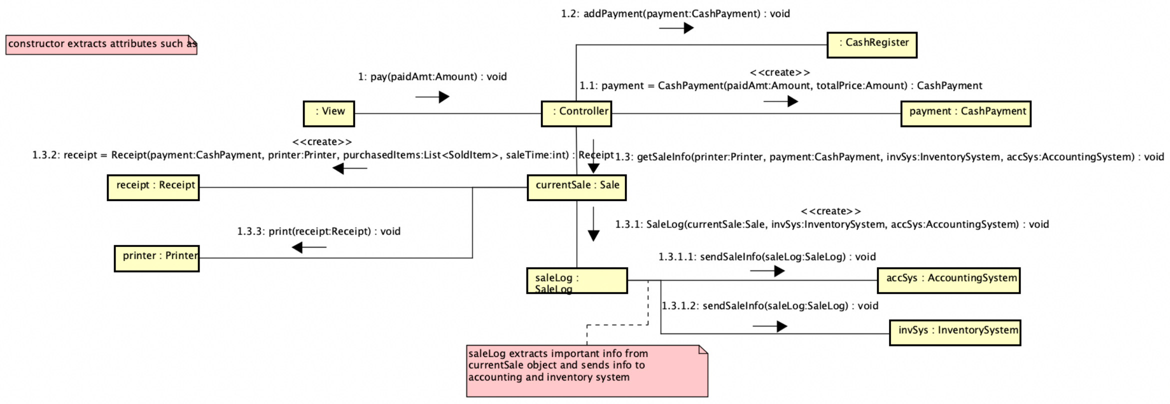

1.2: addPayment(payment:CashPayment) : void
: CashRegister
constructor extracts attributes such as
1: pay(paidAmt:Amount) : void
<<create>>
1.1: payment = CashPayment(paidAmt:Amount, totalPrice:Amount) : CashPayment
: View
: Controller
payment : CashPayment
<<create>>
1.3.2: receipt = Receipt(payment:CashPayment, printer:Printer, purchasedItems:List<SoldItem>, saleTime:int) : Receipt
1.3: getSaleInfo(printer:Printer, payment:CashPayment, invSys:InventorySystem, accSys:AccountingSystem) : void
receipt : Receipt
currentSale : Sale
<<create>>
1.3.1: SaleLog(currentSale:Sale, invSys:InventorySystem, accSys:AccountingSystem) : void
1.3.3: print(receipt:Receipt) : void
printer : Printer
1.3.1.1: sendSaleInfo(saleLog:SaleLog) : void
saleLog : SaleLog
accSys : AccountingSystem
1.3.1.2: sendSaleInfo(saleLog:SaleLog) : void
invSys : InventorySystem
saleLog extracts important info from currentSale object and sends info to accounting and inventory system

Image 4: In this CCD, the total price will be returned to the view after the operation has been executed. The value will be stored to the totalPrice through the model, to then be sent back to the view.

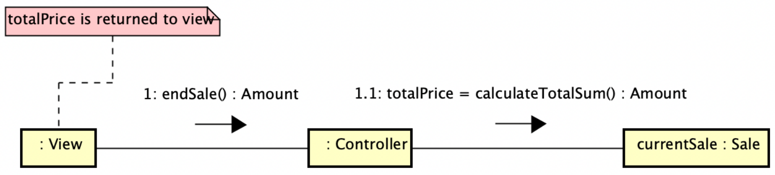

totalPrice is returned to view
1: endSale() : Amount
1.1: totalPrice = calculateTotalSum() : Amount
: View
: Controller
currentSale : Sale

Image 5: CCD diagram for when trying to find the discount code by filling the necessary informations in order to match the customer and retrieve the customer information in the database and the eligible discount.

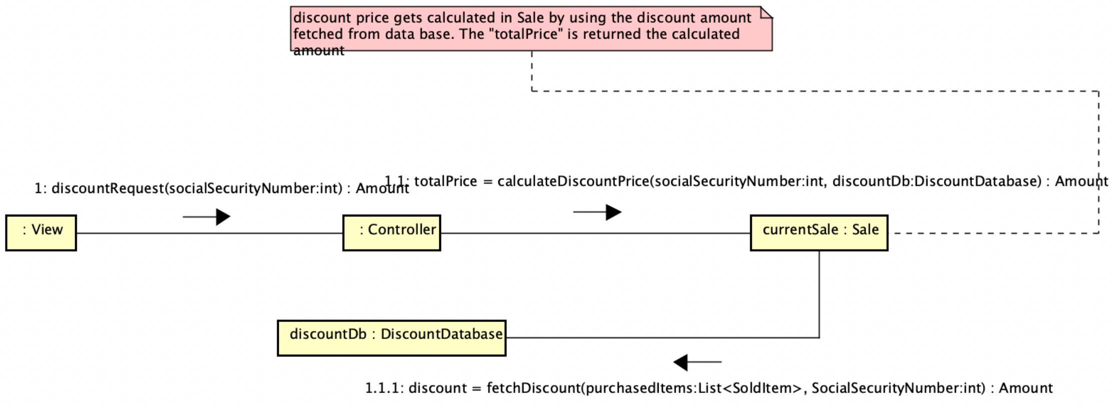

discount price gets calculated in Sale by using the discount amount fetched from data base. The "totalPrice" is returned the calculated amount
1: discountRequest(socialSecurityNumber:int) : Amount
1.1: totalPrice = calculateDiscountPrice(socialSecurityNumber:int, discountDb:DiscountDatabase) : Amount
: View
: Controller
currentSale : Sale
discountDb : DiscountDatabase
1.1.1: discount = fetchDiscount(purchasedItems:List<SoldItem>, SocialSecurityNumber:int) : Amount

Image 6: Represents the result of our MVC and Layers diagram after the classes has been implemented. The arrows shows the relation between the interfaces or specific entities in the interfaces

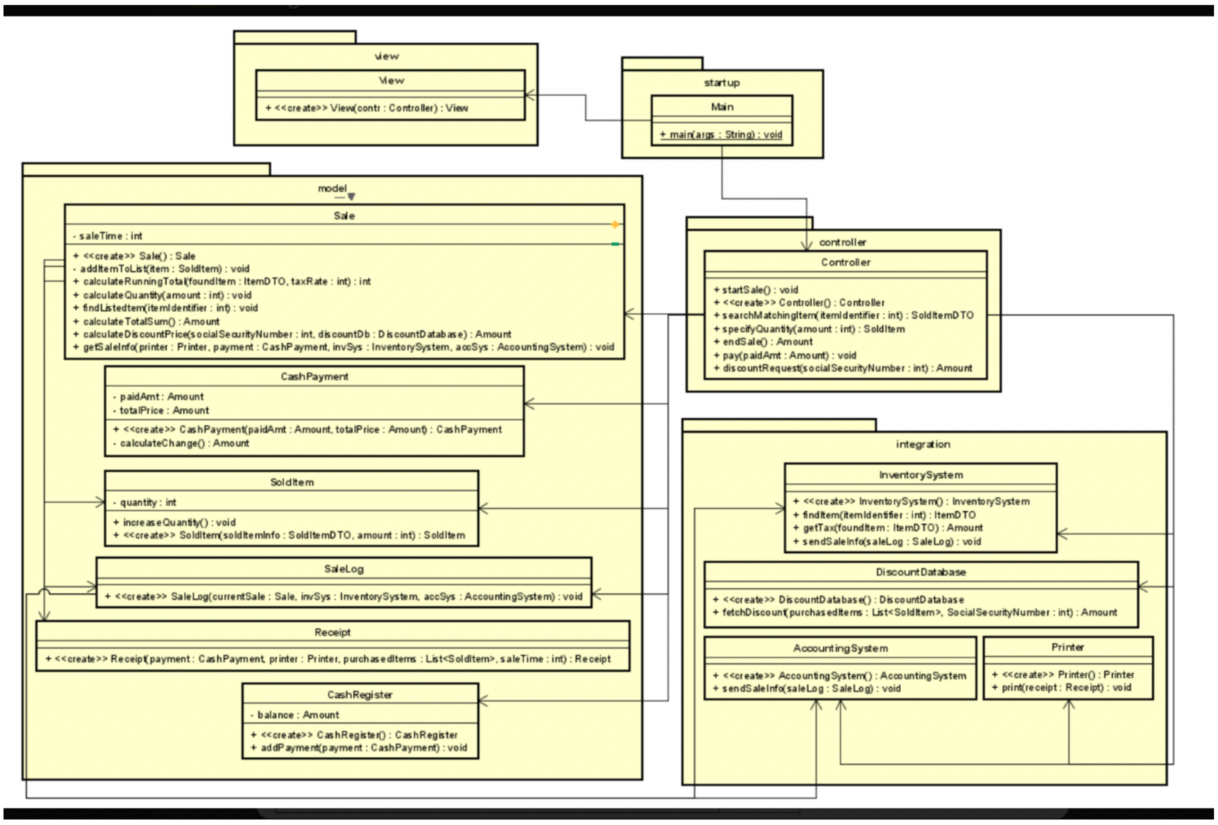

view
View
+ <<create>> View(contr : Controller) : View
startup
Main
+ main(args : String) : void
model
Sale
- saleTime : int
+ <<create>> Sale() : Sale
- addItemToList(item : SoldItem) : void
+ calculateRunningTotal(foundItem : ItemDTO, taxRate : int) : int
+ calculateQuantity(amount : int) : void
+ findListedtem(itemIdentifier : int) : void
+ calculateTotalSum() : Amount
+ calculateDiscountPrice(socialSecurityNumber : int, discountDb : DiscountDatabase) : Amount
+ getSaleInfo(printer : Printer, payment : CashPayment, invSys : InventorySystem, accSys : AccountingSystem) : void
CashPayment
- paidAmt : Amount
- totalPrice : Amount
+ <<create>> CashPayment(paidAmt : Amount, totalPrice : Amount) : CashPayment
- calculateChange() : Amount
SoldItem
- quantity : int
+ increaseQuantity() : void
+ <<create>> SoldItem(soldItemInfo : SoldItemDTO, amount : int) : SoldItem
SaleLog
+ <<create>> SaleLog(currentSale : Sale, invSys : InventorySystem, accSys : AccountingSystem) : void
Receipt
+ <<create>> Receipt(payment : CashPayment, printer : Printer, purchasedItems : List<SoldItem>, saleTime : int) : Receipt
CashRegister
- balance : Amount
+ <<create>> CashRegister() : CashRegister
+ addPayment(payment : CashPayment) : void
controller
Controller
+ startSale() : void
+ <<create>> Controller() : Controller
+ searchMatchingItem(itemIdentifier : int) : SoldItemDTO
+ specifyQuantity(amount : int) : SoldItem
+ endSale() : Amount
+ pay(paidAmt : Amount) : void
+ discountRequest(socialSecurityNumber : int) : Amount
integration
InventorySystem
+ <<create>> InventorySystem() : InventorySystem
+ findItem(itemIdentifier : int) : ItemDTO
+ getTax(foundItem : ItemDTO) : Amount
+ sendSaleInfo(saleLog : SaleLog) : void
DiscountDatabase
+ <<create>> DiscountDatabase() : DiscountDatabase
+ fetchDiscount(purchasedItems : List<SoldItem>, SocialSecurityNumber : int) : Amount
AccountingSystem
+ <<create>> AccountingSystem() : AccountingSystem
+ sendSaleInfo(saleLog : SaleLog) : void
Printer
+ <<create>> Printer() : Printer
+ print(receipt : Receipt) : void

Image 7: Here we are defining the CCD for the statement in that case when the found item has been found. If the item has been found by its given data, the next operation will be executed, hence the operation is going in a sequential order after the statement 1.1 has been found true.

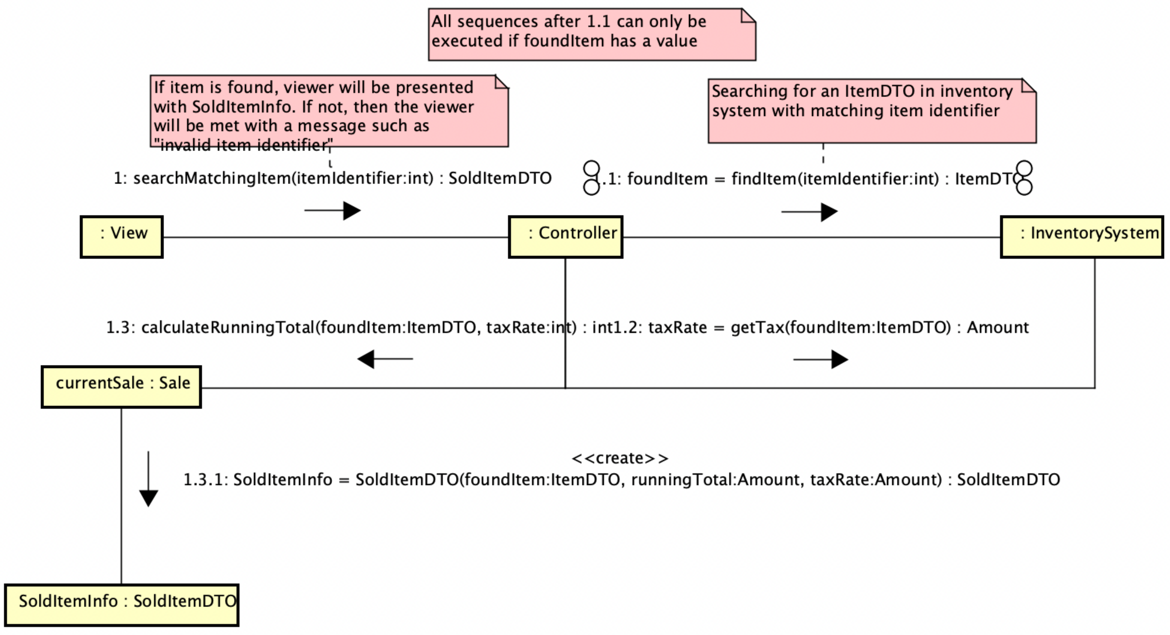

All sequences after 1.1 can only be executed if foundItem has a value
If item is found, viewer will be presented with SoldItemInfo. If not, then the viewer will be met with a message such as "invalid item identifier"
Searching for an ItemDTO in inventory system with matching item identifier
1: searchMatchingItem(itemIdentifier:int) : SoldItemDTO
1.1: foundItem = findItem(itemIdentifier:int) : ItemDTO
: View
: Controller
: InventorySystem
1.3: calculateRunningTotal(foundItem:ItemDTO, taxRate:int) : int
1.2: taxRate = getTax(foundItem:ItemDTO) : Amount
currentSale : Sale
<<create>>
1.3.1: SoldItemInfo = SoldItemDTO(foundItem:ItemDTO, runningTotal:Amount, taxRate:Amount) : SoldItemDTO
SoldItemInfo : SoldItemDTO

Image 8: This diagram shows how the entities interact with each other when the goods are purchased and the program has sold the items.

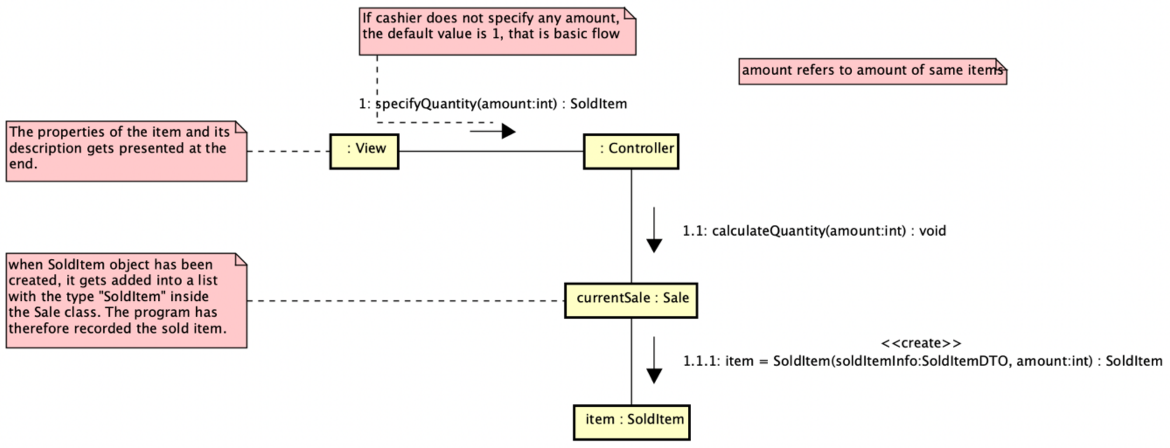
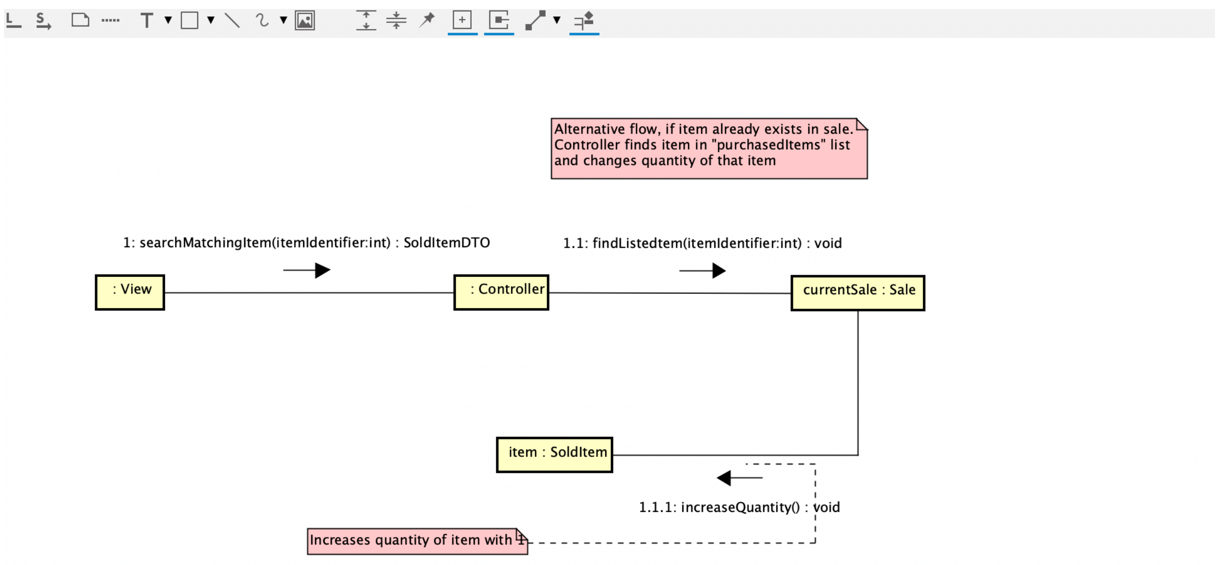

If cashier does not specify any amount, the default value is 1, that is basic flow
amount refers to amount of same items
1: specifyQuantity(amount:int) : SoldItem
The properties of the item and its description gets presented at the end.
: View
: Controller
1.1: calculateQuantity(amount:int) : void
when SoldItem object has been created, it gets added into a list with the type "SoldItem" inside the Sale class. The program has therefore recorded the sold item.
currentSale : Sale
<<create>>
1.1.1: item = SoldItem(soldItemInfo:SoldItemDTO, amount:int) : SoldItem
item : SoldItem

Image 9: This image shows a separate CCD for the alternative flow in that case which the item already exists in sale. Controller finds the item in purchased items list, and the quantity changes for that item in question.


Alternative flow, if item already exists in sale.
Controller finds item in "purchasedItems" list
and changes quantity of that item
1: searchMatchingItem(itemIdentifier:int) : SoldItemDTO
1.1: findListedtem(itemIdentifier:int) : void
: View
: Controller
currentSale : Sale
item : SoldItem
1.1.1: increaseQuantity() : void
Increases quantity of item with 1

5.Discussion

This chapter of the report covers the evaluation of the solutions according to the assessment criteria that is found in the seminar tasks description in Canvas.

Whether if the design is easy to understand would vary depending on if the observer has any previous experiences in reading codes. In the beginning when I firstly came across the communication diagram in our 2nd exercise lecture in prep for seminar 2, it was a real struggle to understand even the smallest interaction that was depicted. As a person without any prior practical background in any programming language, it was difficult to understand the messages and requests that was exchanged during the interaction. After reading the entire chapter of the introduction to the design patterns, looking at the implementation examples and making sure that the design meets the requirements in chapter 5, I can confidently say that the designs are easy enough to understand. The fact that the team consists of people with different levels of programming background has been the greatest advantage because we could be both making sure to develop a complex design that is simultaneously easy understood. One thing that I personally think has made the designs easy to understand, was when we divided every system operations from the SSD to separate CCDs.

This has helped us breaking down the bigger problems into smaller ones by observing how the entities from the different public interfaces in the MVC and Layers design interact with each other for every different and sequential operation in the CCD. During the design implementation of the CCD, we noticed that the design was easily understood by both briefing through the general given requirements specification from task 1 and closely observing how the packages will be communicating with each other in the SSD while creating the CCD. When that curtain operation is executed, how will the request be received to the Model from the Controller after it has received it from the View?

Also, one of the many things that we have learned from task 1 is to keep the design model as easy and simple as we can get while still meeting the !NO criteria from the chapter(s), since it can get too complicated pretty fast. An example of such case is found in the SSD in the last operation pay(amount) when the operation creates a sequence of multiple operations.

Regarding the second criteria, I think that the models has been designed according to their pattern by being attentive to each of the packages defined responsibilities, requirements, and overall architecture of the system. After that, we started off by building a communication diagram for each operation, breaking the bigger system into smaller components. We know that we have enough classes in a package, because we made sure that each class handles a single responsibility, while the layers should serve a well defined purpose. **Give a reference to the diagram as an example.** We also made sure that the controller does not handle any logic except for the model, with an understanding that the controller only has the responsibility to act as an intermediary between the View and the Model. An example of such case is when the controller acts as an input validation, before passing on further of the logic data to the model to be performed with the logic operations. The input validation can be if the input is of the right format. An example of such from the package is the startSale () : void – method signature. By having method signatures defined in the controller package as public, it also serves the purpose to define the available methods in the system and how they can be called.

According to the third criteria, I would say that we have tried modeling our system by considering that each operation achieves the criteria of having low coupling, high cohesion of the data thanks to considering when to change the visibility of curtain methods (for example) to private when they are executed. By following the design pattern that is defined for each of the packages, we can make sure that the interfaces between the components are clear therefore intacted. We also limited the coupling between the interfaces by making sure that the implementation details are encapsulated in interfaces that are not well-defined. An example in such operation are a method operation with a private visibility: - calculateChange (): amount in the cash Payment class, in the model package, where the "-" sign clarifies the visibility. By it being private, it says that the private method can't be called outside the class and is limited within the class it is defined in. A example of a high cohesion is actually the sale class in our method, which after defining every necessary method that is needed seems to be have more methods than most of the classes, which in this case can contribute to a high cohesion.
According to our fifth criteria, I agree with that we have defined the parameters, return values and types correctly as specified, with taking inspiration from the examples in chapter 5. Also by attentively creating the corresponding system operation and following the format for a message exchange in CCD. Also, by following the Java naming conventions for example when defining a method, can remind us whilst writing the operation, to make sure that we have define if the method returns anything. This further on reminds us to put a return value in case we have missed it.