Seminarium 2

Objekt-Orienterad Design, IV1350

Daniel Westerlund
daweste@kth.se

2019-06-01

Innehållsförteckning

1 Introduktion

Seminariet syfte var att lära sig att skapa ett designa ett program som kan hantera alla delar av ett försäljningsscenario (samma kravspecifikation som för seminarium 1). Designen ska ha hög ”cohesion”, låg ”coupling” samt ha en bra inkapsling utav data och ett bra designat publikt gränssnitt. Detta skall designas med hjälp av MVC och lager mönster (View kan ersättas med en klass View).

Jag har arbetat själv.

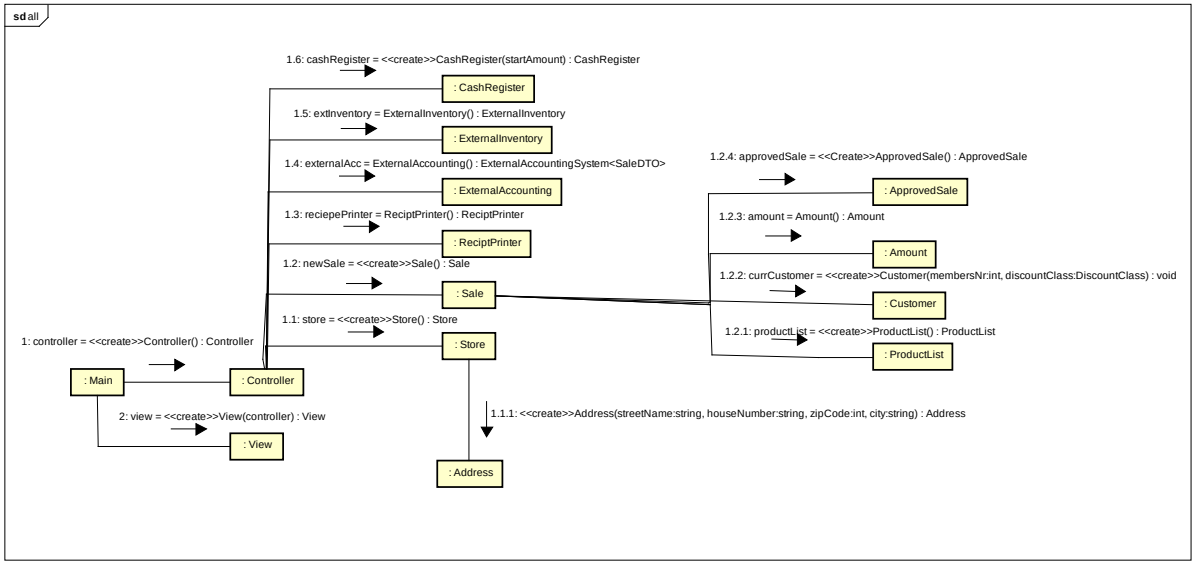
2 Metod

Efter seminarium 1 så justerades både domänmodellen och SDD.

Jag började med att skissa upp en ungefärligt klassdiagram, och därefter påbörjades designen utav interaktionsdiagrammen. Använde SSD som beskriver ordningen på hur allt sker mellan säljaren och systemet, och skapade en systemoperation i taget, som t.ex. addProduct, och sen tänker ett steg till, vilka funktioner/klasser behövs skapas för att kunna lägga till en vara, och därefter lags systemoperationen in den på själva kartan över alla operationer, samt adderade de metoder som kommer att behövas till klassdiagrammet. Därefter valdes nästa operation ut och samma procedur upprepas till att allt ifrån SSD och dess underliggande metoder för att lösa de anropen skapats. Kartan över alla systemoperationen trimmades genom att bara behålla stora drag av vad som händer.

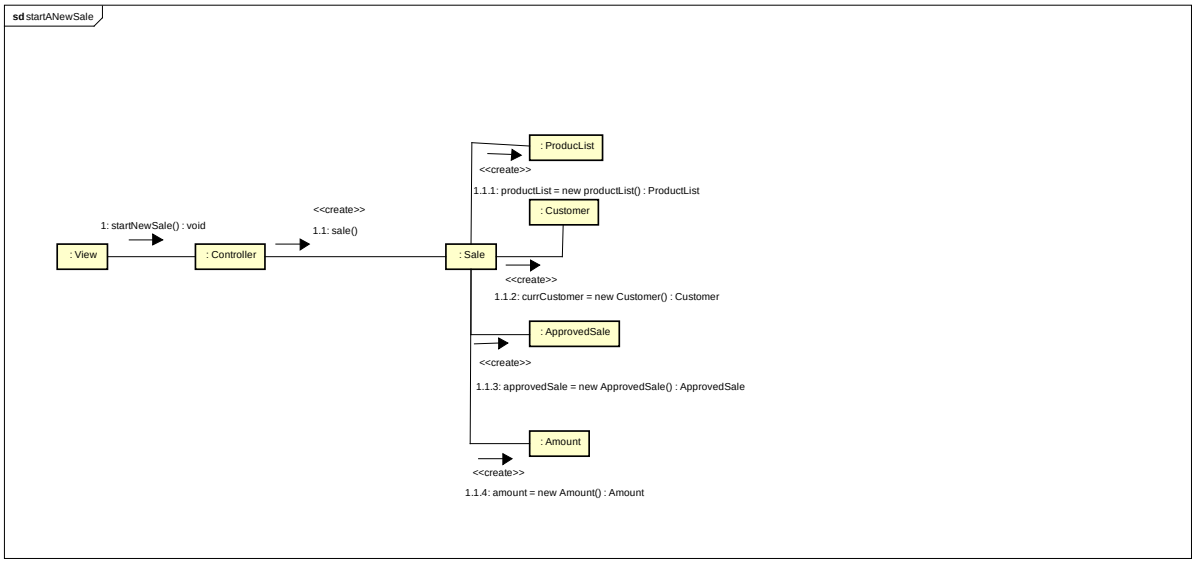
3 Resultat

Figur 1 visar en översikts bild över kommunikation diagrammen i programmet. P.ga. platsbrist är den kraftigt reducerad.



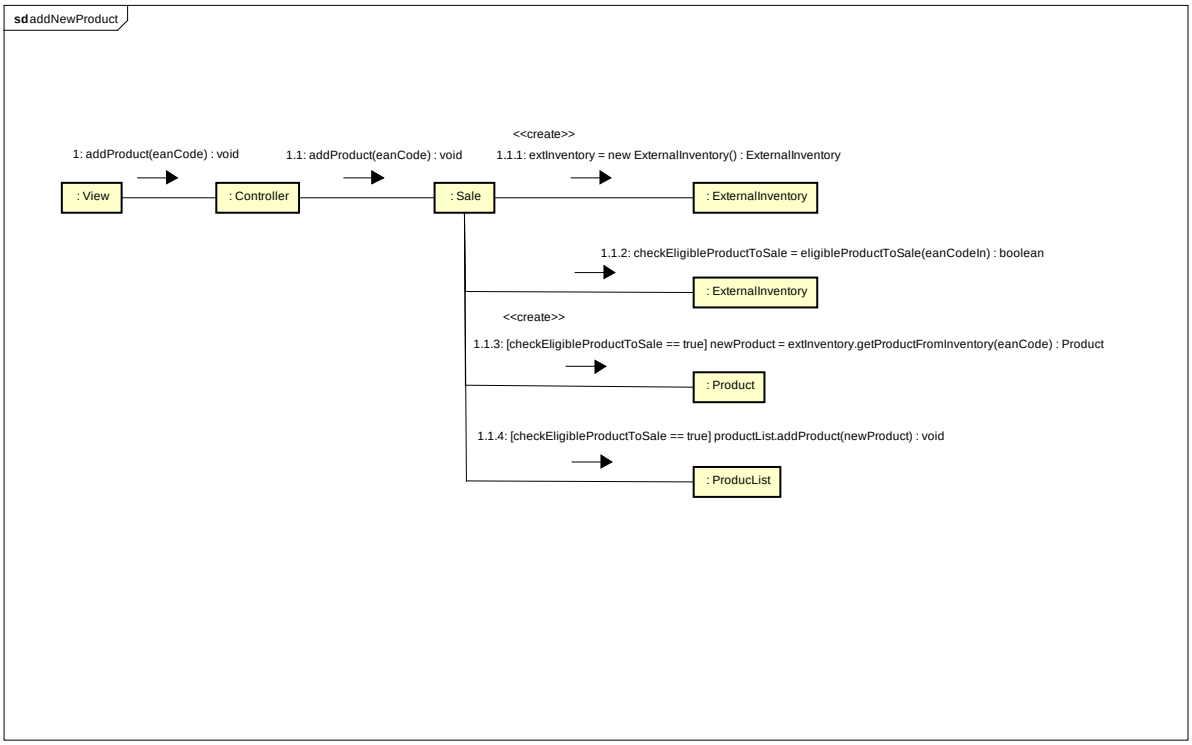
Figur 1- Översiktsbild på kommunikation diagrammen.

Figur 2 visar kommunikation diagrammet för att starta en ny försäljning. View anopar Controller.newSale(), som skapar ett Sale() objekt, som i sin tur skapar alla det objekt som behövs.



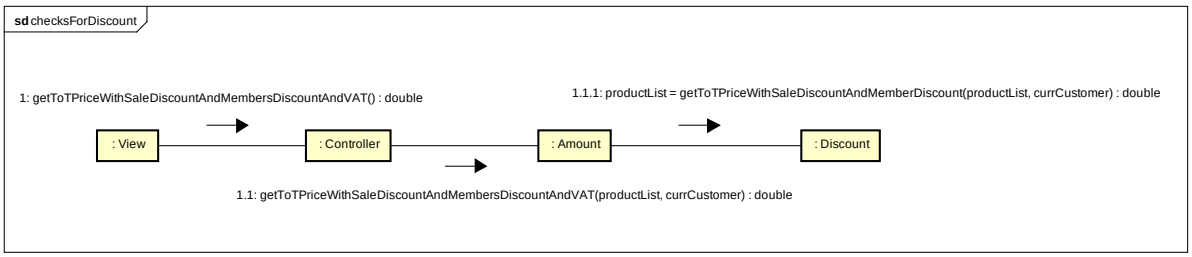
Figur 2- Visar kommunikation diagrammet för startANewSale()

Figur 3 visar kommunikation diagrammet för att addera en produkt till försäljningen. View anropar Controller.addProduct(eanCode). Där parametern eanCode är ett heltal som försäljaren läser av från produkten alternativt har ett externt program som skannar av produkter efter ean-koder. Controllern anropar sale.addProduct(eanCode), som skapar ett ExternalInventory-objekt och kontrollerar om det är en giltigt ean-kod. Därefter hämtas och lagras ett product-objekt i klassen ProductList, som innehåller en ArrayList<Product> med alla produkter i nuvarande köp.



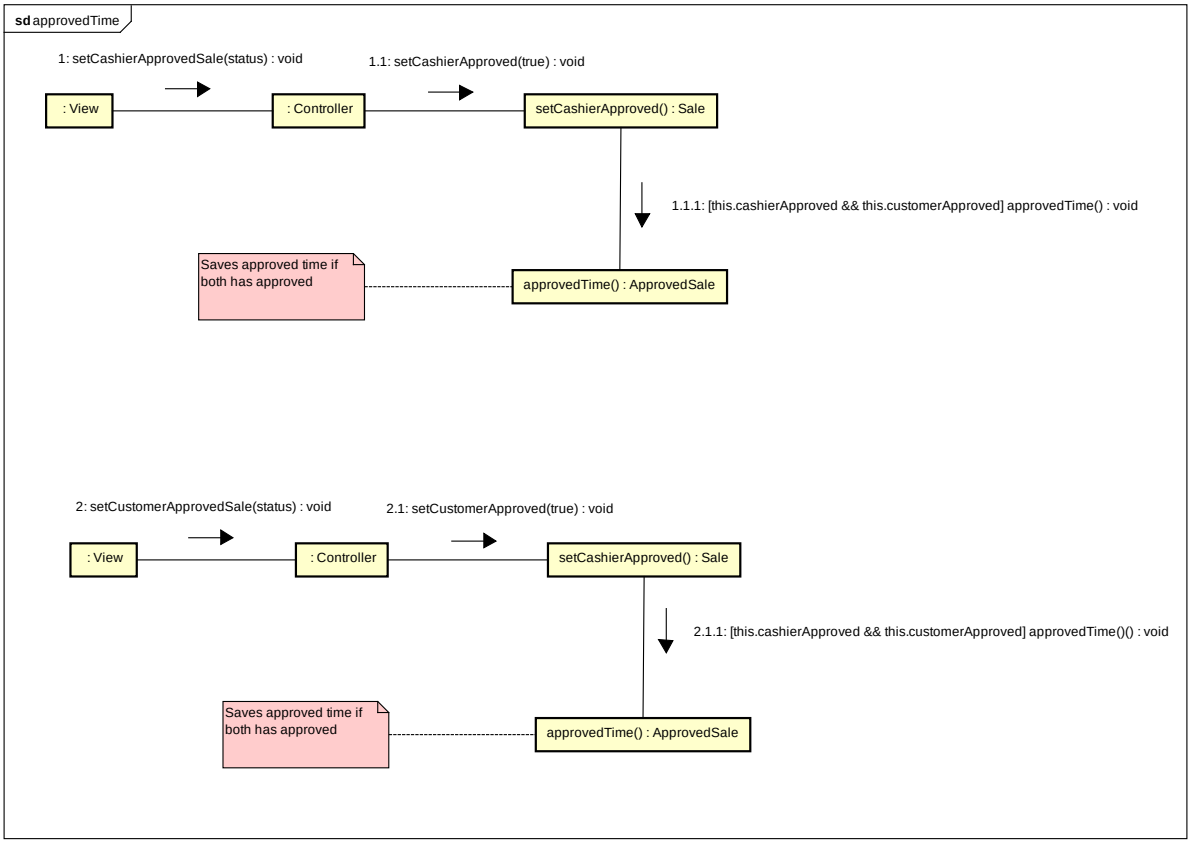
Figur 3- Visar kommunikation diagrammet för lägga till en produkt i nuvarande försäljning.

Figur 4 visar kommunikation diagrammet för att kontrollera rabatt på köpet. View anropar controller. getToTPriceWithSaleDiscountAndMembersDiscountAndVAT som i sin tur Amount. getToTPriceWithSaleDiscountAndMembersDiscountAndVAT(productList, currCustomer), där parametrarna productList innehåller alla produkter i köpet och currCustomer är nuvarande kund. Amount anropar i sin tur Discount. getToTPriceWithSaleDiscountAndMembersDiscountAndVAT(productList, currCustomer) som ansvarar för att alla rabatter läggs på köpet och retunerar ett productList-objekt med uppdaterade rabatter.



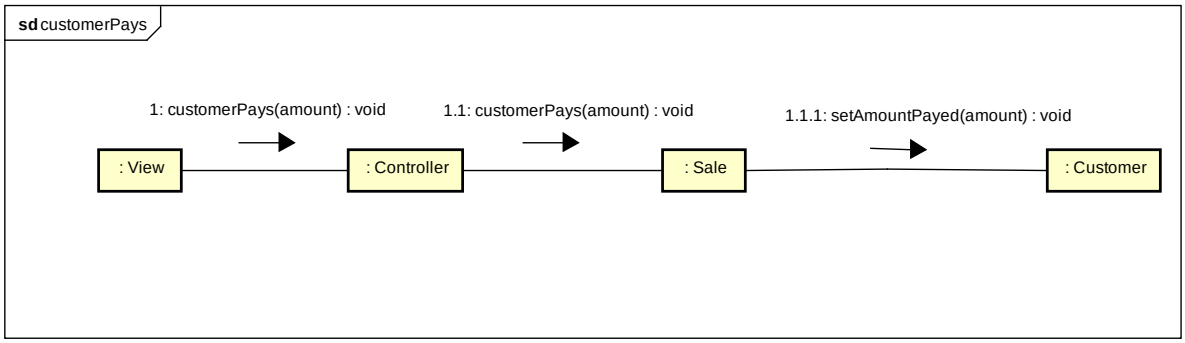
Figur 4 - Visar kommunikation diagrammet för att kontrollera rabatter.

Figur 5 visar kommunikation diagrammet för när både kunden och säljaren godkänner köpet, dvs båda är överens om antal varor och det totala priset. Det är samma procedur för båda (endast säljare accepterar beskrivs) två, där View anropar Controller.setCashierSale(Status). Därefter kontrolleras om båda två har accepterat och tidpunkten sparas ned.



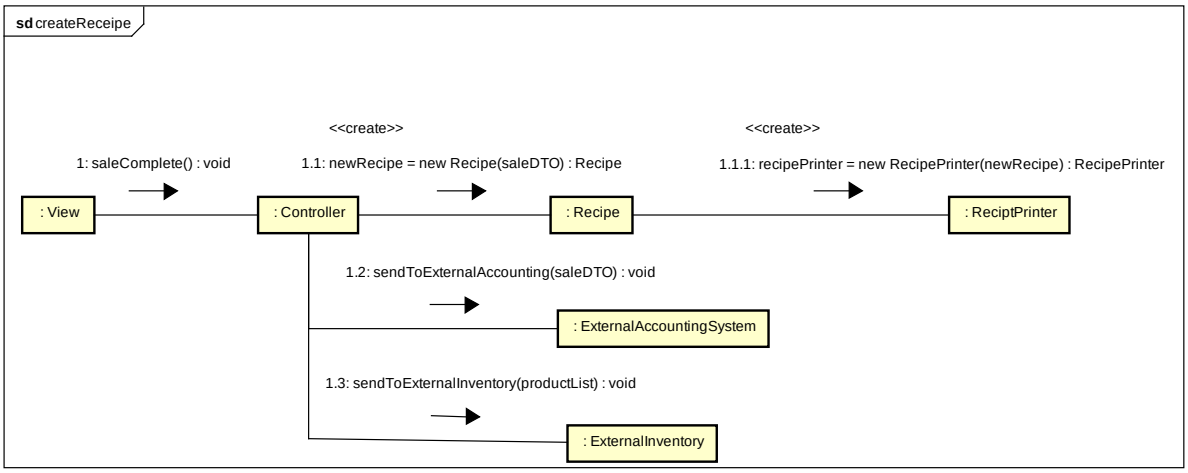
Figur 5 - Visar kommunikation diagrammet för kunden och säljaren accepterar köp.

Figur 6 visar kommunikation diagrammet när en kund betalar. View anropar Controller.customerPays(amount) som sedan anropar Sale.customerPays(amount) dä amount är hur mycket som kunden betalar. Där efter anropar Sale-objektet Customer.setAmountPayed(amout) som tilldelar Customer amountPayed-variabel rätt värde.



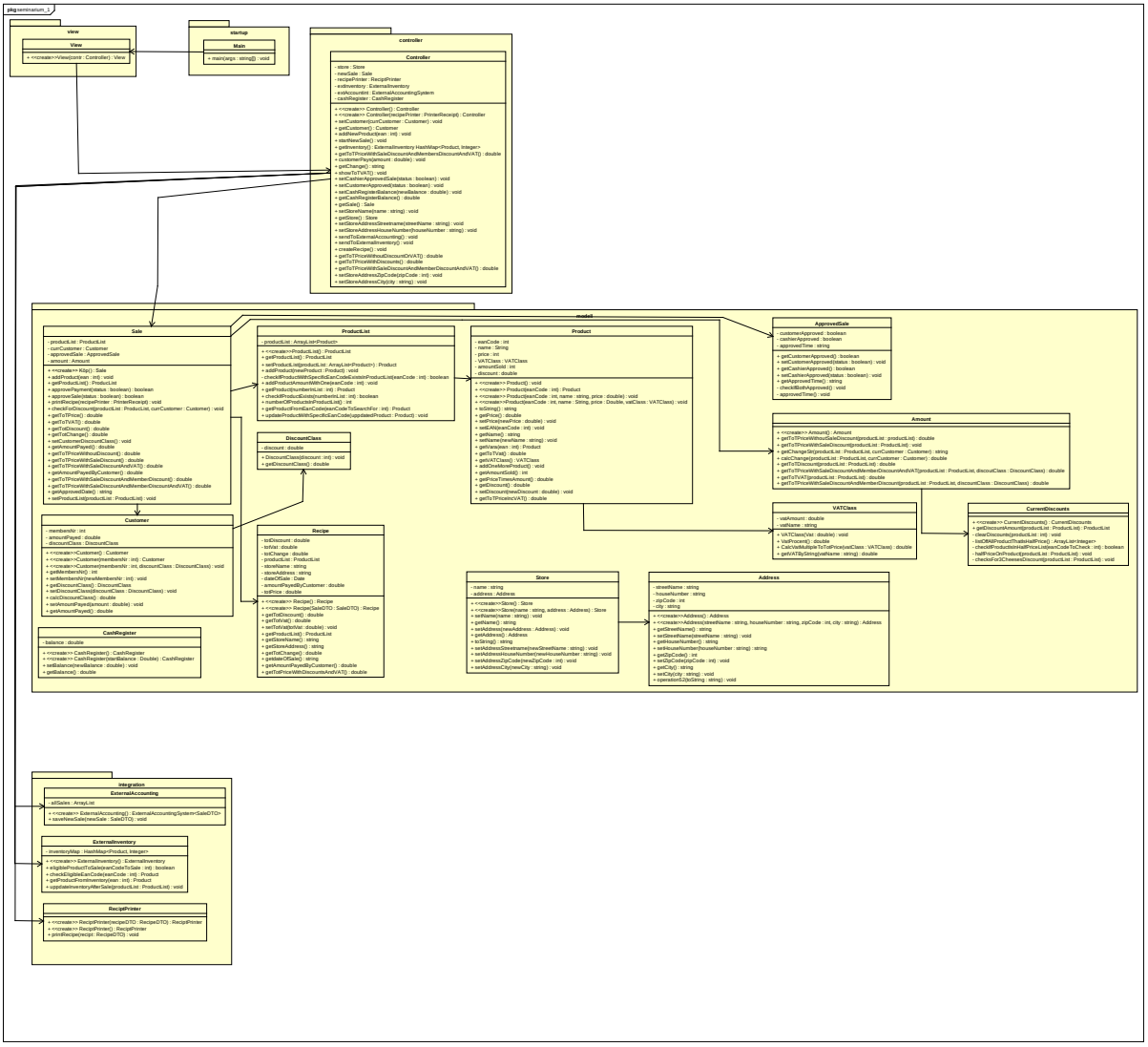
Figur 6 - Visar kommunikation diagrammet för när en kund betalar.

Figur 7 visar kommunikation diagrammet för att skapa kvitto samt att skicka vidare information till externa bokföringsprogrammet och uppdatera lagret. View anropar Controller.saleComplete(), som skapar ett nytt Receipt-objekt baserad på nuvarande köp. Därefter skapas och skickas objekt med rätta data till de externa systemen.



Figur 7- Visar kommunikation diagrammet för att skapa kvitto samt skicka data till de externa systemen.

Relationerna mellan MVC och dess innehåll.



Figur 8 - Visar MVC-schemat över programmet

4 Diskussion

Hade en del problem att komma igång, och veta exakt hur jag skulle lägga upp arbetet. Initialt tyckte jag att det var lättare att börja med klassdiagrammet, förmodligen för att jag har mera erfarenhet utav det. Det vart lite stökigare på det sättet, men jag ser nu varför det är rimligare att börja åt andra hållet, och nu när man har gjort det, kommer det troligtvis vara lättare att börja ”från rätt” håll. Antar att detta är som med domänmodellen och SSD, att man blir bättre ju flera man skapar. Inkapslingen försökte jag lösa med att setter/getters så att när man ska ändra data på objekt ändrar man inte direkt på ”den riktiga datan”. Försökte få ner antalet publika metoder för att begränsa det publika gränssnittet, men just nu ser jag inte flera som jag kan göra om till private, men kanske upptäcker senare i programmeringsdelen, även fast det vore bra om det är löst redan nu. Det är dock lite högre ”coupling” än vad som kanske är rimligt för ett sånt här litet problem. Just nu är det ett litet problem men har man ett större så skulle detta kunna bli problematiskt rätt fort om det fortsätter så här. Även fast jag nog är mera lagd åt det så kallade ”happy-hacking” hållet så har jag insett att det är viktigt och förmodligen extremt tidsparande om man har en bra design, då kommer kodningensdelen att gå fort.

Översiktsbilden försökte jag minska så mycket som möjligt och bara ha den som en enkel överskådsbild över systemet då jag upplever att det vart enklare för mig (och förhoppningsvis andra) att tyda den, och få en bra överblick av hur systemet på fungerar på ett ungefär. Storleksmässigt tycker jag den vart bra, kanske kunde inkluderat något mera, men tycker det är lätt hänt att börja tänka, ”om den metoden ska vara med, borde inte där här också vara med”, och sen slutar det med att överskådsbilden är fullsmetad och man kan inte tyda någonting.

Discount-klassen som har reda på alla rabatter är jag inte riktigt nöjd med, eftersom detta är en reviderad version så har jag i princip gjort klart seminarium 3, dvs programmeringsdelen av program. Anser att Discount-klassen nästan skulle kunna vara ett helt eget system, då den i en verklig miljö skulle bli oerhört stor, just nu har jag den hårdkodad med rabatter men att få den klassen flexibel och lätt att lägga till/ta bort erbjudanden och diverse rabatter kommer göra den klassen väldigt komplicerad.

Som vanligt när man känner sig nöjd och stänger ned diagrammen för att sedan öppna dem någon dag senare, hittar man massor med fel, och det känns verkligen inte som de perfekta diagrammen inte finns.